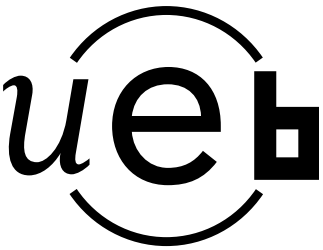
Blocs nuls dans la hiérarchie mémoire

Julien Dusser

N° d'ordre : 4305 ANNÉE 2010



THÈSE / UNIVERSITÉ DE RENNES I
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES I

Mention : Informatique

École doctorale Matisse

présentée par

Julien DUSSER

préparée à l'unité de recherche IRISA, UMR6074
Institut de Recherche en Informatique et Systèmes Aléatoires
IFSIC

Blocs nuls dans la hiérarchie mémoire

Thèse soutenue à Rennes
le 16 décembre 2010

devant le jury composé de :

Olivier SENTIEYS / président
Professeur à l'Université de Rennes I,
ENSSAT – Lannion

Pascal SAINRAT / rapporteur
Professeur à l'Université de Toulouse III,
IRIT – Toulouse

Jean-Michel MULLER / examinateur
Directeur de recherches au CNRS,
LIP, École Normale Supérieure de Lyon

André SEZNEC / directeur de thèse
Directeur de recherche à INRIA,
INRIA Rennes – Bretagne Atlantique

membre invité :

Isabelle PUAUT
Professeur à l'Université de Rennes I,
IRISA – Rennes

Hofstadter's Law : It always takes longer than you expect,
even when you take into account Hofstadter's Law.

Douglas Hofstadter [49]

Remerciements

Je remercie Olivier Sentieys, professeur à l'Université de Rennes 1, de m'avoir fait l'honneur de présider ce jury.

J'exprime ma profonde gratitude à Pascal Sainrat, professeur à l'Université Paul Sabatier et Olivier Temam, directeur de recherche à l'INRIA, pour avoir accepté d'être les rapporteurs de cette thèse.

Je tiens également à remercier Jean-Michel Muller, directeur de recherches au CNRS, et Isabelle Puaut, professeur à l'Université de Rennes 1, d'avoir participé à mon jury.

Je remercie André Seznec pour m'avoir accueilli au sein des équipes CAPS puis ALF et pour avoir dirigé ma thèse tout au long de ces quatre années. Je lui suis profondément reconnaissant pour son attention, sa disponibilité, ses conseils avisés et sa vision éclairée du futur de l'architecture des processeurs.

Je remercie également Éric, Erven et Pierre pour les discussions passionnées à la cafète, ainsi que Benjamin, Damien, David, Guillaume, Jean-François, Junjie, Nathanaël, Ricardo, Thomas et tous les autres membres passés et présents des équipes ALF et CAPS.

Je remercie ma famille ainsi que mes amis qui m'ont toujours apporté leur soutien durant cette thèse.

Enfin, je remercie de tout cœur Bénédicte pour son soutien, ses relectures, les nombreuses kɔ.ʁɛk.sjɔ̃ ɔʁ.tɔ.gʁa.fik ainsi que pour l'organisation du pot.

Table des matières

Introduction

Le processeur et la mémoire sont deux composants intimement liés qui représentent le cœur des ordinateurs. Ce duo est apparu en 1945 avec le premier ordinateur programmable. Depuis, chacun de ces composants a considérablement évolué, nécessitant des changements radicaux dans la façon de les interconnecter.

La première grande évolution touchant le lien entre le processeur et la mémoire, est due à l'augmentation de la fréquence. Cette augmentation a créé un véritable fossé entre le nombre de cycles nécessaires pour accéder à une donnée stockée dans un registre et pour accéder à une donnée stockée en mémoire principale. En effet, au début des années 1980, un accès mémoire ne nécessitait qu'une poignée de cycles. De nos jours, le temps d'accès à la mémoire est beaucoup plus court mais il n'a pas réduit dans les mêmes proportions que le temps de cycle des processeurs. Ainsi, la durée d'un accès mémoire se compte désormais en centaines de cycles. Les mémoires cache sur un puis plusieurs niveaux, ainsi que l'exécution dans le désordre, ont alors vu le jour. Les mémoires cache permettent de stocker près du processeur de petites quantités de données, et donc de diminuer leur la latence d'accès. Cependant, le temps moyen d'accès demeurant toujours trop important, l'introduction de l'exécution dans le désordre a permis de masquer une partie de la latence d'accès. En effet, lors de l'attente d'un accès à la mémoire, l'exécution dans le désordre permet d'exécuter d'autres instructions. Le principal rôle de la hiérarchie mémoire consiste alors à diminuer la latence d'accès.

La deuxième grande évolution touchant particulièrement le lien entre le processeur et la mémoire est due à l'augmentation du nombre de cœurs sur un composant. En effet, depuis le début des années 2000, devant l'impossibilité d'augmenter la fréquence au delà de 4 GHz, les constructeurs ont choisi d'augmenter le nombre de cœurs. Cette augmentation se traduit par un partage d'une partie de la hiérarchie mémoire entre différents cœurs d'exécution. Généralement, le premier niveau de cache est privé, c'est-à-dire réservé à un cœur. Mais, plus l'on descend dans la hiérarchie mémoire, plus le nombre de cœurs partageant un niveau est important. Cela conduit à une saturation de la bande passante entre les différents niveaux de la hiérarchie mémoire. Cette saturation est renforcée par la gestion simultanée de plusieurs processus par cœur (Simultaneous MultiThreading, SMT). Dans un futur proche, si le nombre de cœurs continue à augmenter, la bande passante mémoire va devenir le principal goulot d'étranglement. Dans

ce contexte, un rôle majeur de la hiérarchie mémoire consiste à filtrer les accès afin de diminuer leur nombre et d'éviter ainsi une saturation complète de la bande passante mémoire.

Dans cette thèse, nous proposons des mécanismes permettant une gestion efficace de la hiérarchie mémoire, réduisant la latence d'accès, l'occupation du bus mémoire et augmentant sa capacité. La compression des données transitant dans la hiérarchie mémoire est la solution que nous avons retenue. En effet, de nombreuses études ont montré que les données traversant la hiérarchie mémoire sont fortement compressibles. Nos propres constatations montrent que les blocs de données complètement nuls représentent la majeure partie des données compressibles. Nous proposons donc des modifications peu coûteuses de l'architecture de la hiérarchie mémoire permettant d'exploiter la présence de ces blocs.

Contributions

Dans ce document, nous analysons les données contenues dans les différents niveaux de la hiérarchie mémoire, et nous constatons la présence d'un nombre important de blocs de la taille d'une ligne de cache complètement nuls, et ce pour de nombreuses applications. Nous montrons que la présence de ces blocs nuls ne résulte pas de simples phénomènes d'initialisation, mais que les blocs nuls sont utilisés tout au long de l'exécution de ces applications. La présence de ces blocs nuls nous permet de faire deux propositions d'architectures. Notre première proposition se situe au niveau du dernier niveau de cache et la seconde au niveau de la mémoire principale.

Notre première contribution est le *Zero-Content Augmented Cache*. Il s'agit d'une modification du dernier niveau de cache permettant d'exploiter le nombre important de blocs nuls présents en mémoire. Nous proposons d'utiliser un cache spécialisé dans le stockage des blocs nuls conjointement avec le cache traditionnel. Ce cache, appelé cache de zéros, permet non seulement de stocker à moindre coût une grande quantité de blocs nuls, mais aussi de libérer de la place dans le cache principal. Un bloc nul est représenté dans le cache de zéros par un seul bit. Ainsi, quelques kilo-octets de stockage physique seulement permettent de stocker plusieurs méga-octets de blocs nuls.

Notre seconde contribution est la *Decoupled Zero-Compressed Memory*. Il s'agit d'une proposition de compression matérielle de la mémoire principale dans laquelle seuls les blocs nuls sont compressés. Cette architecture de mémoire compressée permet de réduire significativement la taille de la mémoire physique nécessaire pour faire tenir le *working-set* de nombreuses applications. Cette proposition utilisée simultanément avec un *Zero-Content Augmented Cache* permet une diminution importante du nombre d'accès mémoire et une augmentation significative des performances.

Organisation du document

Ce document de thèse se compose de quatre chapitres : un état de l'art, puis une analyse de l'utilisation des blocs nuls au sein de la hiérarchie mémoire, suivie de notre proposition de cache spécialisé pour stocker les blocs nuls, et enfin notre proposition de mémoire compressée exploitant elle aussi la présence de blocs nuls.

L'état de l'art présente la compression de données au sein de la hiérarchie mémoire. Dans ce chapitre, nous commençons par décrire le fonctionnement des algorithmes de compression les plus fréquemment utilisés. Ensuite, nous abordons la compression de données dans la hiérarchie mémoire par une étude des propositions de cache de donnés compressées. À l'issue de cette étude, nous descendons dans la hiérarchie mémoire pour examiner les différentes propositions de mémoires principales compressées : la compression logicielle, la compression matérielle et, à mi-chemin entre les deux, la compression de données au sein des machines virtuelles Java.

Le deuxième chapitre consiste en une analyse détaillée de la présence de blocs nuls au sein de la hiérarchie mémoire. En effet, la présence d'une proportion significative de blocs nuls tout au de l'exécution de l'application peut au premier abord surprendre. Nous mesurons donc de façon approfondie le taux de blocs nuls dans les accès aux différents niveaux de la hiérarchie mémoire. Dans une deuxième partie, nous évaluons l'évolution de ce taux au cours de l'exécution de leurs utilisations afin de nous assurer que ces blocs nuls ne sont pas de simples phénomènes d'initialisation. Enfin, dans une troisième partie, nous montrons pourquoi des applications lisent et écrivent des blocs nuls à travers l'étude de leurs codes sources.

Le troisième chapitre présente notre première contribution architecturale : le *Zero-Content Augmented Cache*. Il s'agit d'une proposition de modification du dernier niveau de cache afin de stocker efficacement les blocs nuls. Nous présentons successivement sa structure, son fonctionnement et une analyse détaillée des performances. À l'issue de cette évaluation, nous proposons une modification permettant de réduire drastiquement le surcoût de notre proposition.

Le quatrième, et dernier, chapitre présente notre deuxième contribution : la *Decoupled Zero-Compressed Memory*. Celle-ci permet de compresser la mémoire principale en ne stockant pas les blocs nuls. Dans ce chapitre, nous évaluons le taux de blocs nuls présents dans la mémoire principale. Ensuite, nous présentons la structure et le fonctionnement de la *Decoupled Zero-Compressed Memory*. Pour finir, nous proposons de modifier cette mémoire afin de considérer un algorithme de compression plus complexe que la simple compression des blocs nuls.

Chapitre 1

État de l'art de la compression de données au sein de la hiérarchie mémoire

Depuis les années 1990, de nombreuses études ont montré que les données présentes dans la hiérarchie mémoire sont fortement compressibles. Il a par exemple été observé pour de nombreuses applications que plus de la moitié des octets présents en mémoire sont des zéros. De nombreuses propositions d'architectures visant à compresser les données au sein de la hiérarchie mémoire ont alors été proposées.

Dans ce chapitre, nous allons décrire les différents travaux relatifs à la compression de données dans la hiérarchie mémoire. Dans un premier temps, nous présenterons rapidement le fonctionnement des algorithmes de compression de données sans perte les plus fréquemment rencontrés. Dans un deuxième temps, nous explorerons les différentes propositions de mémoires compressées. Et pour finir, nous nous intéresserons à la compression de la mémoire principale, qu'elle soit matérielle ou logicielle.

1.1 Algorithmes de compression de données sans perte

Il existe deux principaux types de compression de données : la compression sans perte et la compression avec pertes. Seule la compression sans perte est envisageable pour une hiérarchie mémoire. En effet, le changement d'un seul bit dans la zone de code d'une application suffit à changer une instruction en une autre pouvant provoquer ainsi une sortie en erreur ou un résultat incorrect. Par exemple, en assembleur x86, l'*opcode* `0x74` est un `je` alors que `0x75` est l'instruction contraire, c'est-à-dire un `jne`. Le remplacement de l'un par l'autre change radicalement l'exécution.

Dans cette section, nous allons présenter succinctement deux types principaux d'algorithmes de compression sans perte : les algorithmes de codage entropique et les algorithmes de compression à dictionnaire.

1.1.1 Codage d'Huffman

Les premiers véritables algorithmes de compression sont publiés en 1948 et 1952. Il s'agit d'algorithmes de codage. Les premiers algorithmes sont le *codage de Shannon-Fano* [83, 37], et *codage de Huffman* [51]. Contrairement au *codage de Shannon-Fano*, le *codage de Huffman* est un codage optimal, c'est-à-dire qu'il n'existe pas de modification de ce codage permettant d'obtenir un message plus court. Depuis, des dizaines d'autres algorithmes ont vu le jour, chacun adapté à des contraintes particulières ou à un format de données spécifique.

Dans le *codage de Huffman*, la donnée d'entrée est découpée en symboles. Le codage consiste à affecter les codes les plus courts aux symboles les plus fréquents. Une table de correspondance symbole $\mapsto$ code est obtenue en construisant un arbre binaire à partir des fréquences des différents symboles. La donnée compressée est obtenue en remplaçant chaque symbole par le code correspondant. La table inverse est utilisée pour la décompression.

Si le *codage de Huffman* est appliqué sur des données brutes, le taux de compression obtenu est généralement faible. En effet, l'efficacité des algorithmes de codage dépendent fortement du format des données à compresser. Ils nécessitent de définir une méthode permettant de découper la donnée en mots, et d'établir des statistiques de fréquences des différents mots.

Le *codage de Huffman* est généralement utilisé dans des méthodes de compression hybrides, après une transformation avec ou sans perte. Cette transformation permet d'obtenir un format régulier, dont on peut ensuite extraire des mots qui possèdent une répartition non-aléatoire. Le codage permet alors d'exploiter cette répartition. Les formats de compression de fichiers *gzip* et *bzip2* sont conçus de cette façon. *gzip* utilise une compression par dictionnaire (LZ77) suivie d'un *codage de Huffman*. Quant à *bzip2*, il utilise le *codage de Huffman* après une transformée de *Burrows-Wheeler* [20].

En raison de la nécessité de transformer et d'examiner préalablement les données, les algorithmes de codage sont peu utilisés dans la compression de données dans la hiérarchie mémoire. Ils sont plutôt adaptés à une compression hors-ligne.

1.1.2 LZ77 et ses variantes

L'algorithme LZ77 proposé en 1977 par *Lempel et Ziv* est un algorithme de compression à dictionnaire. Le dictionnaire est construit dans une fenêtre coulissante qui se déplace sur les données à compresser. Le dictionnaire contient alors les dernières données rencontrées.

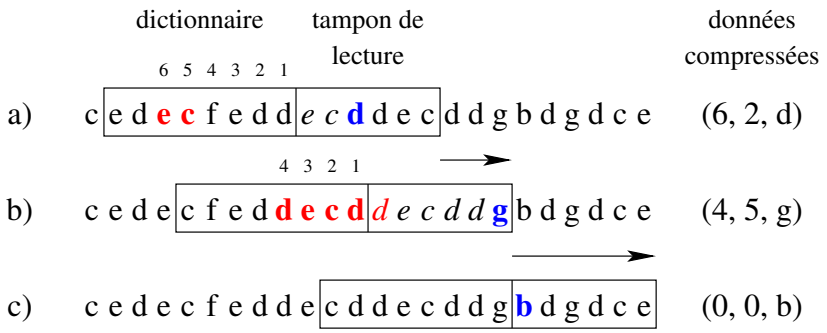


FIGURE 1.1 – LZ77

La fenêtre coulissante est composée de deux parties : le dictionnaire et le tampon de lecture. Lorsqu'une suite de caractères du tampon de lecture est compressée, elle est recherchée dans le dictionnaire. La donnée compressée est composée d'un triplet comprenant la position dans le dictionnaire, la longueur de la chaîne trouvée et le premier caractère différent.

L'exemple de la figure 1.1 illustre le fonctionnement de *LZ77*. À l'étape a, la chaîne `ecddec` est recherchée dans le dictionnaire. Le motif de *deux* caractères `ec` est trouvé en position *six*. La sortie compressée est donc (6, 2, `d`). La fenêtre est décalée de trois positions. À l'étape b, la chaîne `decddg` est recherchée. Le motif de *cinq* caractères `decdd` est trouvé en position *quatre*, sachant que le motif peut être partiellement récursif. La sortie compressée est alors (4, 5, `g`). La fenêtre est décalée de six positions. A l'étape c, la chaîne `bdgdce` est recherchée. Aucune correspondance n'est trouvée. La sortie est alors (0, 0, `b`).

Cette dernière étape de l'exemple illustre bien l'un des points faibles de *LZ77* : l'encodage des nouvelles valeurs. Elles sont en effet représentées par le triplet (0, 0, `x`), qui occupe beaucoup plus d'espace que le caractère ajouté. *LZSS* [85] est une variante de *LZ77* qui permet de ne pas stocker le triplet dans ce cas de figure. Un marqueur permet de désactiver la compression sur un passage.

LZ77 est à l'origine de multiples travaux dérivés. Parmi les plus utilisés on peut citer *LZ78* [105] et sa variante *LZW* [93]. Ces deux algorithmes utilisent un dictionnaire implicite. Contrairement à LZ77, le dictionnaire est créé d'après les données à venir et non pas d'après les données déjà compressées. En 1991, *Ross Williams* a proposé *LZRW* [94]. Il s'agit d'une variante très optimisée, plus simple et utilisant moins de mémoire. Elle permet une compression en 13 instructions par octet et une décompression en 4 instructions par octet. De multiples autres variantes existent, parmi lesquelles : *LZJB*, *LZMA*, *LZO*, *LZT*, *LZWL*, *LZX* et *ROLZ*. Ces variantes permettent de répondre à diverses contraintes de vitesse de compression (ou de décompression), de complexité ou d'efficacité.

Quelque soit la variante, les algorithmes de type *LZ77* présentent tous une efficacité limitée sur les données de petites tailles. En effet, si le bloc de données à compresser

est petit, le dictionnaire a une taille réduite, et peu de motifs vont être retrouvés dans ce dictionnaire. Ces algorithmes nécessitent aussi un temps d'initialisation assez long, le dictionnaire étant initialement vide. Les premiers motifs ne seront alors pas compressés. *Franaszek, Robison et Thomas* [43] ont analysé l'évolution du taux de compression de *LZ77* en fonction de la taille de la donnée. En dessous de 512 octets, le taux de compression est fortement dégradé. La taille minimale idéale semble être comprise entre 512 octets et $1,5$ Ko.

En conclusion, ces algorithmes sont, certes, adaptés à la compression de données dans la hiérarchie mémoire car ils sont rapides, et présentent un bon taux de compression, mais leur efficacité est conditionnée à la taille des blocs de données qui doivent atteindre au moins 1 Ko. Ils sont donc plus adaptés à une compression logicielle des données pour archivage.

1.1.3 Frequent Pattern Compression

L'algorithme de compression *Frequent Pattern Compression* [8] (*FPC*) proposé par *Alameldeen et Wood* en 2004 est également un algorithme à dictionnaire. Cependant, contrairement au *codage de Huffman*, il utilise un dictionnaire statique prédéfini qu'il n'est donc pas nécessaire de stocker. Pour leur version destinée à la compression dans le cache L2, *Alameldeen et Wood* proposent d'utiliser un dictionnaire à huit motifs pour représenter des mots de 32 bits.

Avant compression		Après compression		
Motif	Valeur	Préfixe	Valeur	Taille
mot nul	`00000000`	000	–	3
mot de 4 bits	`SSSSSSSa`	001	`a`	7
mot de 8 bits signé	`SSSSSSab`	010	`ab`	11
mot de 16 bits signé	`SSSSabcd`	011	`abcd`	19
mot de 16 bits	`0000abcd`	100	`abcd`	19
2 mots de 8 bits signés	`SSabSScd`	101	`abcd`	19
4 mots de 8 bits répétés	`abababab`	110	`ab`	11
mot non-compressé	`abcdefgh`	111	`abcdefgh`	35

Table 1.1 – FPC à huit motifs. Les champs *valeur* sont notés en hexadécimal. S est l'extension du bit de signe (soit 0x0, soit 0xF).

L'entrée est parcourue par mot de 32 bits. Chacun de ces mots est recherché parmi les motifs et est remplacé par le couple (préfixe, valeur) correspondant. Les motifs doivent être adaptés à l'*endianness* de la machine.

Ainsi, par exemple, la chaîne hexadécimale de 16 octets `00000085`, `00000000`, `ffffffc5`, `0bfffff0` est compressée en quatre paires (011, `0085`), (000, –), (010, `c5`), (111, `0bfffff0`). Elle occupe alors $19 + 3 + 11 + 35 = 68$ bits.

Avant compression		Après compression		
Motif	Valeur	Préfixe	Valeur	Taille
mot nul	`00000000`	00	–	2
mot de 8 bits	`000000ab`	01	`ab`	10
mot de 16 bits signé	`SSSSabcd`	10	`abcd`	18
mot non-compressé	`abcdefgh`	11	`abcdefgh`	34

Table 1.2 – FPC à quatre motifs proposé par *Ekman et Stenström*

Ekman et Stenström [34] ont mesuré les performances de l'algorithme *FPC* pour la compression de la mémoire principale. Ils ont observé que les motifs les plus importants étaient ceux correspondant aux mots nuls et partiellement nuls. Dans leurs mesures, une compression *FPC* simplifiée avec quatre motifs offrait un taux de compression très proche d'un *FPC* à huit motifs. Ils ont montré que *FPC* permet d'obtenir un taux de compression [1] d'environ 50% pour des blocs de 64 octets.

1.1.4 X-Match et ses variantes

L'algorithme de compression X-Match a été proposé en 1996 par *Kjelsø, Gooch et Jones* [56]. Comme *LZ77*, cet algorithme construit un dictionnaire au fur et à mesure de la compression. Une nouvelle entrée est ajoutée au dictionnaire pour chaque nouveau mot rencontré.

Lors de la compression, la donnée d'entrée est découpée en mots de quatre octets. Un par un, les mots sont ensuite recherchés dans le dictionnaire. Si une entrée du dictionnaire a au moins deux octets identiques avec le mot recherché, celui-ci est alors considéré comme compressible. Dans le cas contraire, le mot ne sera pas compressé.

Un mot non-compressé est représenté par la paire (1, *mot non-compressé*), il occupera donc 33 bits de stockage.

Un mot compressé est représenté par le quadruplet (0, *position dans le dictionnaire*, *position des caractères trouvés dans le mot*, *caractères restants*). Ce quadruplet occupe entre 5 et 32 bits, selon le nombre de caractères communs entre le mot à compresser et le mot trouvé dans le dictionnaire.

Afin de stocker efficacement les suites de zéros, les auteurs proposent une version légèrement modifiée de l'algorithme, nommée X-RL. Cependant, la complexité matérielle de la compression augmente sérieusement. *Ahn, Yoo et Kang* proposent eux aussi de petites modifications de X-Match. Celles-ci permettent des gains substantiels de compressibilité [6] sans trop augmenter la complexité de l'algorithme.

1. Le taux de compression est défini par $\tau = \frac{Taille\ non\text{-}compressée - Taille\ compressée}{Taille\ non\text{-}compressée}$

Comme la majorité des algorithmes construisant un dictionnaire au fur et à mesure de la compression, cet algorithme est particulièrement adapté aux blocs de données de quelques kilo-octets. Si la taille de bloc descend en dessous de 1 Ko, le taux de compression se dégrade sérieusement.

1.1.5 Élimination des mots nuls

En 1997, *Rizzo* [74] a constaté qu'une grande partie des mots de 32 bits présents en mémoire sont des mots nuls. Il a proposé un mécanisme de compression adapté à la compression des mots nuls au sein d'une page de 4 Ko.

La page est découpée en deux parties, une partie stockant les mots non-nuls et une partie contenant une table de bits (*bitmap*). La table de bits est utilisée pour indiquer pour chaque mot s'il est nul ou non-nul. S'il est non-nul, la valeur est lue dans la première partie de la page.

Après compression, il constate que la partie correspondant à la table de bits, contient elle aussi de nombreux zéros. Il utilise alors le même mécanisme sur la seconde partie de la page, avec une taille de mot de 8 bits. Le gain d'espace supplémentaire ainsi obtenu s'explique par la présence de blocs nuls d'au moins trente-deux octets.

Le taux de compression [2] obtenu ainsi est important mais moins bon que *FPC*, il se situe autour de 40%.

1.2 Compression des caches

Une dizaine de propositions de caches compressés ont été faites [59, 58, 92, 101, 103, 100, 23, 7, 70, 71, 89]. Cependant, la compression des données à l'intérieur du cache pose de nombreuses difficultés : L'efficacité de l'algorithme sur de petits blocs, la latence de décompression et le stockage des blocs compressés.

Parmi les propositions, nous allons décrire les plus avancées, à savoir celles de *Yang, Zhang et Gupta*, puis celle d'*Alameldeen et Wood* et pour finir celle de *Qureshi, Suleman et Patt*.

1.2.1 Propositions de *Yang, Zhang et Gupta*

En 2000 et 2003, *Yang, Zhang et Gupta* ont publié trois propositions différentes de caches compressés [101, 103, 100]. Ils sont partis des observations de Lipasti [64] et Gabbay [46] qui ont remarqué que de nombreuses instructions produisent régulièrement les mêmes valeurs. Ils proposent alors de stocker ces *valeurs fréquentes* dans une table dédiée et de remplacer la valeur par l'index dans la table, l'index étant généralement codé sur trois bits.

Dans la publication [103], *Zhang, Yang et Gupta* proposent de diminuer le nombre d'échecs d'un cache *direct mapped*. Ils suggèrent d'ajouter un *Frequent Value Cache* à

2. cf. note 1 page précédente

coté du cache principal. Le *Frequent Value Cache* est un cache compressé qui ne stocke que les index des mots compressés, les mots non-compressés n'étant pas stockés. Afin de ne pas dégrader les performances sur les mots non-compressés, le *Frequent Value Cache* est utilisé comme un *victim cache*. Lors de l'éjection d'un bloc du cache principal, une entrée est ajoutée au *Frequent Value Cache*. Ainsi, ce dernier ne peut qu'améliorer les performances. Lors d'un succès, l'entrée est invalidée dans le *Frequent Value Cache* et insérée dans le cache principal. De ce fait, la donnée n'est jamais présente simultanément dans les deux caches. Cette proposition permet de diminuer le nombre d'échecs.

Dans la publication [101], *Yang, Zhang et Gupta* proposent une autre architecture de cache compressé. Chaque ligne du cache compressé peut accueillir soit un bloc non-compressé soit deux blocs compressés. Si au moins la moitié des mots d'un bloc sont des *valeurs fréquentes*, il peut être stocké compressé. La demi-ligne stocke alors les valeurs non-fréquentes. L'index des *valeurs fréquentes* est stocké, quant à lui, dans un champs additionnel de trois bit par mot. Cette proposition permet d'augmenter la quantité de données stockées, mais elle a un coût important. Elle nécessite de doubler le nombre de *tags* et aussi d'ajouter un champ de trois bits par mot. La latence de décompression qui risque d'avoir un impact sur les performances n'est pas mesurée.

Dans la publication [100], *Yang et Gupta* proposent d'utiliser les *valeurs fréquentes* pour diminuer la consommation électrique du cache. Pour les valeurs fréquentes, seuls le tag et quatre bits sont lus au lieu du tag et trente-deux bits. Pour les *valeurs non-fréquentes*, la latence d'accès est augmentée d'un cycle.

1.2.2 Proposition de *Alameldeen et Wood*

Alameldeen et Wood ont mesuré que lors de l'utilisation d'un cache compressé avec FPC, les performances peuvent aussi bien augmenter de 18% que diminuer de 17% selon l'application [7]. Cette dégradation est due à la latence de décompression qui n'est pas toujours compensée par le gain d'espace obtenu dans le cache. Ce sont principalement les applications tenant dans le cache qui sont concernées. Afin de résoudre ce problème, *Alameldeen et Wood* proposent en 2004 d'utiliser un mécanisme adaptatif [7, 98].

La compression est réalisée à l'aide d'un cache associatif par ensemble de huit voies mais qui ne dispose d'un espace de stockage que de quatre voies. Ainsi, pour chaque *set* le cache contient huit *tags*, mais seulement quatre à huit blocs selon leurs compressibilités.

Un prédicteur est utilisé pour déterminer l'efficacité de la compression. La position de la ligne dans la LRU permet de détecter un succès dû à la compression. En effet, si un bloc placé après la quatrième position provoque un succès, il y aurait eu un échec sur ce bloc sans compression. Si le nombre de succès dus à la compression ne permet pas de couvrir la latence de décompression, le prédicteur peut alors choisir de ne pas compresser un bloc. Dans ce cas, les *tags* sont quand même mis à jour pour continuer à évaluer l'efficacité de la compression. La dégradation des performances est alors limitée à $0,4\%$.

1.2.3 Filtrage des mots d'une ligne de cache

Les caches utilisent la localité spatiale des accès. Ainsi, si un mot a été accédé, les autres mots de la même ligne ont alors une forte probabilité d'être accédés eux aussi. Cependant, pour certaines applications, la localité spatiale n'est pas toujours réelle. Ce phénomène est d'autant plus présent que la ligne est proche de la position LRU [71]. En 2007, deux propositions visant à filtrer les blocs non-utilisés ont été faites [71, 70]. *Qureshi, Suleman et Patt* proposent d'utiliser pour chaque *set* la voie du L2 destinée à stocker la ligne LRU afin de stocker les mots utiles de plusieurs blocs. *Pujara et Aggarwal* proposent, quant à eux, d'effectuer le filtrage sur toutes les lignes du cache de données de niveau 1.

La compression est obtenu en stockant des mots issus d'autres lignes dans l'espace libéré par le filtrage. Aucune décompression n'est nécessaire : il n'y a donc pas d'impact sur la latence d'accès. Les difficultés principales viennent de l'augmentation de la complexité de la structure du cache, des *tags* supplémentaires à gérer, et de la prédiction des mots non-utilisés.

1.2.4 Zero-Value Cache d'Islam et Stenström

Cette proposition de Zero-Value Cache [54] a été publiée simultanément avec notre Zero-Content Augmented Cache qui sera détaillée au chapitre 3. Nos deux propositions suggèrent d'utiliser un cache spécialisé dans le stockage de données nulles. Cependant, la taille des blocs nuls considérés et la structure du cache sont radicalement différents. Néanmoins, cela montre l'intérêt de traiter efficacement les blocs nuls au sein de la hiérarchie mémoire.

En 2009, *Islam et Stenström* ont constaté qu'une part non-négligeable des *loads* accèdent a des mots nuls [54]. Ils ont mesuré sur les SPEC CPU 2000 un taux de mots nuls compris entre 5 et 59% avec une moyenne à 18%. Ces mesures sont effectuées avec SimpleScalar, SimPoint et un jeu d'instructions Alpha. Nos propres mesures effectuées avec Pin et un jeu d'instructions x86 montrent un taux plus faible, compris entre 5 et 28% avec une moyenne proche de 9%.

Le *Zero-Value Cache* proposé par *Islam et Stenström* vise le premier niveau de la hiérarchie mémoire. Un petit cache est utilisé afin de traquer les octets nuls dernièrement accédés. Ainsi, lors d'un *load* le cache est accédé, si tous les octets du mot lu sont nuls alors le *Zero-Value Cache* peut répondre à la requête.

La difficulté principale de la gestion d'un cache de premier niveau est la granularité variable des accès à une même donnée. En effet, les processeurs modernes proposent plusieurs instructions *load* et *store* de 1, 2, 4 ou 8 octets. Il est donc nécessaire de garder un bit par octet et non un bit par mot afin de gérer les données nulles.

1.3 Compression sur le bus mémoire

Afin d'améliorer la bande passante mémoire, quelques propositions de compression des données transitant sur le bus mémoire ont été faites [26, 38, 21, 55, 14, 25]. Les adresses et les données sont compressées avant d'être envoyées sur le bus et décompressées dès la réception. La majorité des ces propositions utilisent le fait que seuls les bits de poids faibles ont une forte entropie. Les bits de poids fort sont, quant à eux, souvent identiques.

La première proposition est celle de *Citron et Rudolph* [26], publiée en 1995. Ils proposent d'utiliser un dictionnaire pour les bits de poids forts afin de diminuer la largeur du bus. À chaque extrémité du bus, un dictionnaire est maintenu d'après les dernières valeurs transférées. Lors d'un transfert, si une entrée de même valeur que les bits de poids forts à transférer est trouvée, alors seuls l'index dans le dictionnaire et les bits de poids faibles sont transférés. Sinon, la donnée est transférée en plusieurs cycles. Ainsi, ils ont mesuré que pour 90% des transferts un bus de 16 bits utilisant ce mécanisme de compression permet de transférer en un cycle autant de données qu'un bus de 32 bits.

1.4 Compression de la mémoire principale

Les propositions de compression de la mémoire principale peuvent être classées en trois catégories : logicielles, matérielles et machines virtuelles.

1.4.1 Compression logicielle

On dénombre de nombreuses propositions utilisant une compression logicielle sur la mémoire principale. *Wilson* en 1991 [95] est le premier à suggérer de compresser les pages non-récemment utilisées. En 1993, *Douglis* propose une structure complète de mémoire compressée [30]. Ensuite, de nombreuses autres propositions ont suivi. En décembre 2009, *compcache* [27], un mécanisme de mémoire compressée a été intégré dans la version officielle du noyau Linux 2.6.33.

1.4.1.1 Structure

Toutes les propositions utilisant une compression logicielle de la mémoire ont la même structure. La figure 1.2 page suivante présente cette structure. Une partie de la mémoire, appelée cache compressé, est réservée afin de stocker les pages compressées. Seul ce cache est compressé, le reste de la mémoire n'est pas modifié. Lors d'un accès à une donnée dans ce cache compressée la page complète est décompressée dans l'espace non-compressé. Le gain de performance obtenu vient du temps de décompression de la

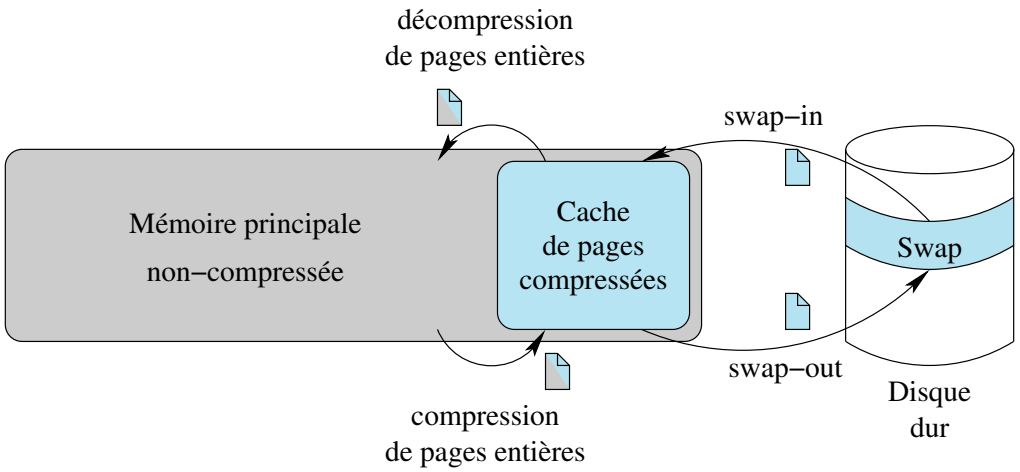


Figure 1.2 – Compression logicielle de la mémoire. Un cache compressé est réservé au sein de la mémoire principale.

page qui est inférieur au temps d'accès au disque dur, et de la forte compressibilité des pages mémoire.

Cependant, la taille du cache compressé est difficile à établir. *Douglis* a constaté qu'un cache trop petit ne permet pas de stocker suffisamment de pages pour être efficace [30]. Inversement, si le cache compressé est trop gros de nombreuses pages qui auraient tenu en mémoire non-compressée vont être stockées dans le cache. Le temps d'accès à ces pages va alors augmenter.

Les propositions de mémoire compressée en logiciel peuvent être classées en deux catégories : celles dont la taille du cache compressé est déterminée statiquement et celles dont la taille du cache compressé s'adapte à son efficacité.

Cache de taille statique

On dénombre assez peu de propositions de compression logicielle de la mémoire avec un cache compressé de taille fixe. Les principales propositions sont [75] et [22].

Roy, Kumar et Prvulovic proposent dans [75] d'utiliser un drivers Linux afin de réaliser un cache compressé. La principale limitation est la difficulté de trouver une taille de cache qui satisfasse toutes les applications.

Cervera, Cortes et Becerra proposent dans [22] de réserver seulement quelques mégaoctets afin de créer un tampon vers un *swap* compressé. Ce tampon permet de regrouper plusieurs pages compressées au sein d'un bloc du disque dur. Ainsi, lors du chargement d'un bloc depuis le disque dur, un mécanisme de *prefetch* apparaît.

Cache de taille adaptative

Les propositions de compression logicielle avec un cache de taille variable sont beaucoup plus nombreuses. On peut principalement citer les publications : [30, 96, 29, 47, 91, 48, 102, 12, 13]

Toutes ces propositions sont fondamentalement très proches. Chacune propose d'utiliser une nouvelle structure de données pour stocker le cache compressé, ou bien une nouvelle heuristique pour déterminer sa taille, ou encore d'utiliser un nouvel algorithme de compression plus rapide.

1.4.1.2 Adaptations du système d'exploitation

Dans les propositions précédentes, l'utilisation d'une mémoire compressée de façon logicielle nécessite une modification assez profonde du système.

Le système d'exploitation doit être adapté pour ne pas éjecter les pages vers le périphérique de stockage de masse, mais plutôt vers le cache compressé en mémoire. De plus, ce cache doit aussi être géré : sa taille doit être contrôlée et des pages compressées doivent être éjectées vers le périphérique de stockage de masse. Tous les prototypes reposent sur un noyau Linux modifié.

1.4.2 Compression matérielle

Contrairement à la compression logicielle de la mémoire, la compression matérielle permet de compresser toute la mémoire, et non pas seulement un cache dans lequel les pages sont stockées compressées. Ainsi, le processeur accède directement aux données compressées sans qu'il y ait besoin de les recopier préalablement dans une zone non-compressée.

Les contraintes appliquées à l'algorithme de compression ne sont pas les mêmes que pour une compression logicielle de la mémoire. Tous les blocs étant compressés, à chaque accès les blocs de données compressés doivent être décompressés. Il est en conséquence primordial que l'algorithme de compression permette une décompression très rapide. Par contre, l'exigence d'efficacité de l'algorithme est beaucoup plus faible puisqu'il est appliqué sur l'ensemble de la mémoire, et non simplement sur une partie de celle-ci.

Afin de garantir les accès aléatoires, la mémoire est découpée en blocs de taille fixe qui sont ensuite compressés. La taille de ces blocs, une fois compressés, est variable. Cela rend leur stockage difficile. Ce stockage est rendu encore plus complexe par l'évolution possible de la taille des blocs lors d'une réécriture. Cela rend nécessaire un nouveau niveau d'adressage.

Parmi les différentes propositions, deux se distinguent particulièrement : la technologie IBM MXT qui a été commercialisée [90] et une proposition d'*Ekman et Stenström* [34].

1.4.2.1 IBM Memory eXpansion Technologies (MXT)

IBM a proposé au début des années 2000 des serveurs équipés de la technologie MXT [90] permettant de compresser la mémoire principale. Le premier objectif de l'utilisation de cette technologie était de diminuer le prix de vente [84]. En effet, diviser par deux la quantité de mémoire présente permettait de réduire le prix d'un tiers.

Architecture

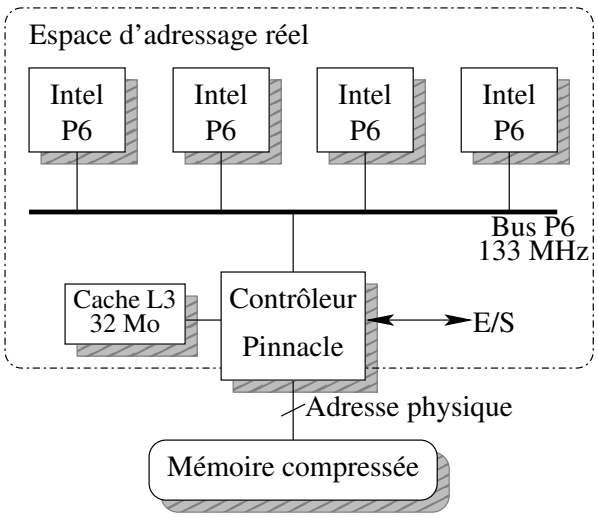


Figure 1.3 – Architecture IBM MXT

La mémoire compressée IBM MXT, décrite dans la figure 1.3, repose sur un contrôleur mémoire nommé *Pinnacle* dont le fonctionnement est décrit dans les publications [41, 90, 39, 42]. Il est capable de gérer l'accès à la mémoire compressée. Deux espaces d'adressage sont définis : l'espace des adresses réelles et l'espace des adresses physiques. La traduction d'adresse se fait dans le contrôleur. Afin de masquer le plus possible les latences de compression et de décompression, un cache L3 de 32 Mo contenant des lignes de 1 Ko est ajouté. Il est composé de mémoire SDRAM DDR, plus rapide que la mémoire principale compressée qui est composée de simple SDRAM [90].

La compression est effectuée dans le contrôleur *Pinnacle* qui intègre une version parallèle [43] de l'algorithme LZ77 [104]. Elle est appliquée à des blocs de 1 Ko qui sont compressés en 0, 1, 2, 3 ou 4 secteurs de 256 octets.

Afin de pouvoir adresser la mémoire compressée, une table de traduction, représentée sur la figure 1.5 page ci-contre, est stockée non-compressée. Elle occupe 16 octets par bloc de 1 Ko. Chaque entrée, représentée sur la figure 1.4 page suivante, est composée d'un champ de contrôle et de quatre pointeurs de 30 bits permettant d'adresser les différents secteurs où est stocké le bloc compressé.

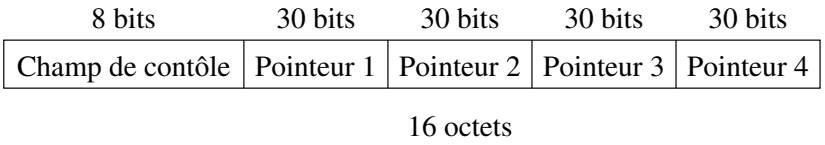


Figure 1.4 – Une entrée de la table de traduction. Elle est composée d'un champ de contrôle et de quatre pointeurs vers les secteurs où est stocké le bloc compressé.

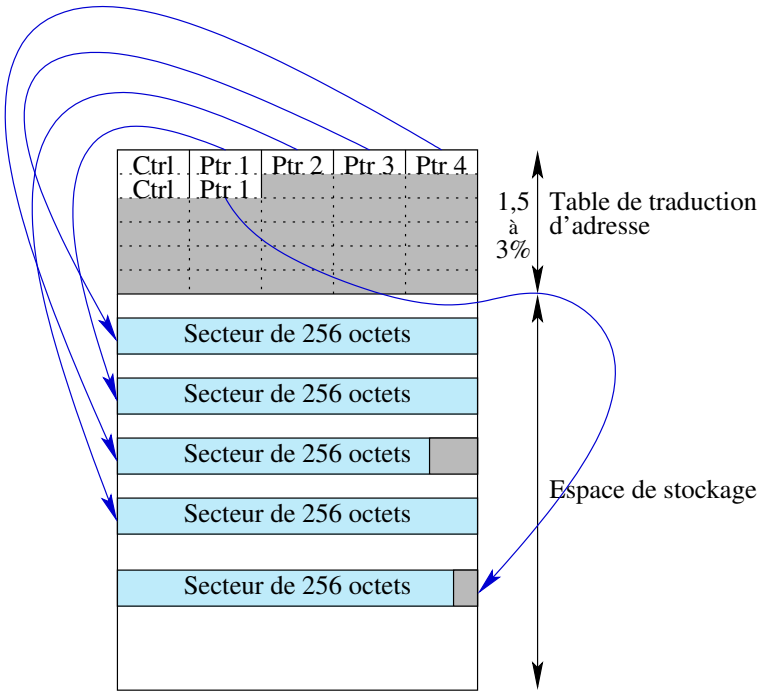


Figure 1.5 – Organisation de la mémoire. Une partie de l'espace sert à stocker les pointeurs de secteurs.

Plus le contenu de la mémoire est compressible, plus elle contient de lignes, et plus il faut agrandir cette table. Cette dernière occupe donc environ $1,5\%$ de la mémoire pour un taux de compression de 0% et $3,1\%$ de la mémoire pour un taux de 50%.

Dans le but d'optimiser l'utilisation de l'espace physique, deux techniques ont été proposées. On peut facilement les comparer à celles actuellement utilisées dans les systèmes de fichiers.

– Si le bloc une fois compressé utilise 120 bits ou moins, il peut être stocké dans les quatre champs réservés aux pointeurs. Sur la figure 1.6 page suivante, la ligne 4 est stockée de cette façon. Aucun secteur n'est donc alloué. Pour comparaison, dans les systèmes de fichiers ext4 et NTFS, les petits attributs de fichiers sont directement stockés respectivement dans les *inodes* [36] et dans la *MFT* [68].

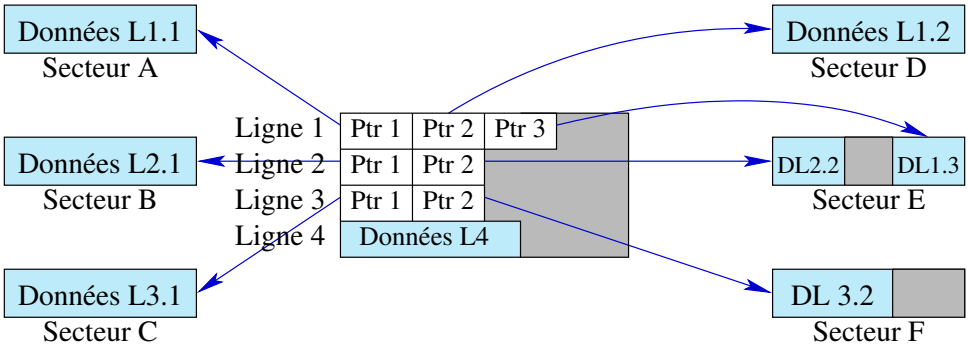


Figure 1.6 – Optimisation de l'espace de stockage. Les lignes 1 et 2 partagent le même secteur E. La ligne 4 est stockée directement dans la table.

– Si le dernier secteur dans lequel est stocké le bloc compressé est rempli à moins de 50%, il peut accueillir la fin d'un autre bloc compressé de la même page. Sur la figure 1.6, c'est le cas pour les lignes 1 et 2. Pour comparaison, cette technique appliquée aux systèmes de fichiers est appelée *Tail packing*. Elle est utilisée notamment par *Reiser4* [72].

Le choix d'une taille de bloc de 1 Ko résulte d'un compromis. Dans l'article présentant la version parallèle de LZ77 de *Franaszek, Robison et Thomas* [43], une analyse de l'évolution du taux de compression en fonction de la taille des blocs est effectuée. En dessous de 512 octets, le taux de compression est fortement dégradé. Au dessus de $1,5$ Ko, le taux de compression est stable. Les blocs doivent donc être suffisamment larges pour obtenir un bon taux de compression avec LZ77 (et LZSS). Inversement, sur la question de l'accès aux données compressées, une taille de bloc trop importante entraîne une grande latence de décompression. La taille choisie est donc un compromis entre le taux de compression et le temps d'accès.

La taille des secteurs fixée à 256 octets résulte également d'un compromis [41, 42]. L'utilisation de petits secteurs permet d'éviter la fragmentation de l'espace disponible. Ils ont cependant un coût de gestion très important, car il faut adresser chacun d'entre eux. En effet, l'utilisation de plus petits secteurs impliquerait plus de secteurs, plus de pointeurs, et des pointeurs plus longs. L'espace réservé à la table de traduction augmenterait donc, réduisant l'espace disponible pour les données compressées. Par ailleurs, de larges structures sont difficiles à faire tenir en cache et augmentent la latence de traduction d'adresse.

Afin d'améliorer le support de l'espace compressé, des modifications sont appliquées au système d'exploitation. Ces modifications se présentent sous la forme d'un driver pour Windows 2000, ou d'une modification du noyau Linux. Elles sont décrites par *Abali et al.* dans [3, 1, 2]. *Franaszek, Heidelberger et Wazlowski* [40] proposent quant à eux une analyse de la gestion de l'espace libre. Le système est initialisé avec une taille de mémoire principale de deux fois la taille de la mémoire physique réellement présente.

La principale modification consiste, lorsque le taux de compression est trop faible, à éjecter des pages vers le périphérique de stockage de masse afin de diminuer la pression sur la mémoire compressée.

Performances

L'analyse de performances réalisée par *Abali et al.* [4] montre que pour certaines applications cette architecture réussit presque à doubler la taille de la mémoire, c'est-à-dire que le *working-set* que peut contenir la mémoire MXT est de deux fois la taille de la mémoire physique. La vitesse d'exécution de six applications parmi les SPEC CPU 2000 s'améliore même de 4 à 8%. Une application de base de données interne à IBM est aussi testée. Elle montre une réduction impressionnante de 66% du temps d'exécution.

Cependant, d'autres applications des SPEC CPU 2000 s'exécutent jusqu'à 10% plus lentement. Aucun détail n'est donné sur l'application de base de données ni sur le jeu de données réellement stockées. Les gains semblent en réalité provenir de la forte compressibilité des données manipulées et des mauvaises performances de l'application avec une mémoire non-compressée. De plus, il s'avère qu'une partie importante des gains de performances vient de l'utilisation d'un cache L3 non-compressé de 32 Mo constitué de mémoire SDRAM DDR plus rapide que la mémoire SDRAM.

Limitations

Le principal facteur limitant de cette architecture est la granularité des accès à la mémoire principale. L'impact des soixante-quatre cycles de latence de décompression des blocs de 1 Ko masque en partie les gains de la compression. Certaines applications accédant à de nombreuses petites structures telles que des listes chaînées risquent aussi de saturer la bande passante mémoire.

1.4.2.2 Proposition d'Ekman et Stenström

Début 2005, *Ekman et Stenström* ont tout d'abord proposé de hiérarchiser la mémoire principale sur deux niveaux [33]. D'après leurs mesures, si le premier niveau occupe 30% de la taille de la mémoire principale et que le deuxième niveau est plus lent d'un facteur dix, l'impact sur les performances n'est que de $1,2\%$. Ainsi, le deuxième niveau plus lent peut être constitué de mémoire moins chère, ou même une mémoire compressée. Cette approche ressemble à l'architecture IBM MXT précédemment décrite avec un cache SDRAM DDR de 32 Mo.

La même année, *Ekman et Stenström* ont fait une proposition de mémoire compressée matériellement [34]. Celle-ci se démarque principalement de l'approche d'IBM par l'utilisation de petits blocs de 64 octets.

Architecture

Ekman et Stenström sont partis de l'observation de *Yang et Gupta* [100] qu'une large partie des pages, des blocs et des mots sont nuls. Les mesures effectuées par *Ekman et Stenström* d'après des instantanés de la mémoire montrent qu'en moyenne 30% des blocs de 64 octets sont complètement nuls, et que 55% des octets sont nuls. Une version simplifiée de l'algorithme FPC proposé par *Alameldeen et Wood* [7, 8] est alors utilisée. Le taux de compression moyen sur les applications choisies est d'environ $0,5$.

Lorsqu'un bloc de données est compressé, quatre tailles S sont considérées : 0, 22, 44 ou 64 octets. Ces tailles sont codées sur deux bits dans une table de traduction d'adresses. Une taille de 0 indique un bloc nul. Le bloc n'est alors pas alloué en mémoire.

La principale complexité de la gestion de blocs de 64 octets est l'impossibilité d'associer un pointeur de 32 bits par bloc ($6,25\%$ de l'espace mémoire serait utilisé pour la table d'indexation). *Ekman et Stenström* proposent de stocker les blocs contigus de façon séquentielle. Ainsi, la position P_x du bloc x peut se déduire d'après les tailles des blocs précédents. $P_x = \sum_{i=0}^{x-1} S_i$. Mais, afin de réduire le nombre d'additions nécessaires au calcul d'adresse, les pages sont divisées en sous-pages dont les tailles sont elles codées sur deux bits. *Ekman et Stenström* proposent d'utiliser quatre tailles de sous-pages : 256, 512, 768 et 1024 octets. La figure 1.7 page ci-contre représente le découpage des pages en sous-pages et blocs.

Le calcul de l'adresse d'un bloc est illustré sur la figure 1.8 page 32. L'adresse correspond à la somme des tailles des sous-pages précédentes et des blocs précédents au sein de la sous-page.

La gestion du changement de la taille d'un bloc compressé est assez délicate. Lorsque la taille d'un bloc compressé augmente et qu'il n'existe pas suffisamment d'espace libre après le bloc, tous les blocs suivants sont décalés, et éventuellement les sous-pages suivantes. La complexité de ce décalage est réellement importante, surtout si le bloc se situe en début de page. De plus, si la page est trop petite, elle doit être déplacée et allouée ailleurs.

Inversement, lorsque la taille d'un bloc compressée diminue, un décalage est également nécessaire. Mais celui-ci peut être interrompu si besoin, en contrepartie d'une fragmentation interne plus importante. Cependant, la fragmentation de l'espace libre au sein des blocs, des sous-pages et des pages permet d'absorber une partie des décalages.

Afin d'accélérer la recherche d'un bloc compressé, la table de traduction d'adresses est mise en cache au niveau du TLB. Cette structure occupe le double de la taille du TLB. Elle est accédée pour tout accès au L2.

Performances

L'utilisation de cette architecture permet de réduire en moyenne de 30% l'espace mémoire occupé par les applications simulées. Ce taux de compression de 30%, est donc

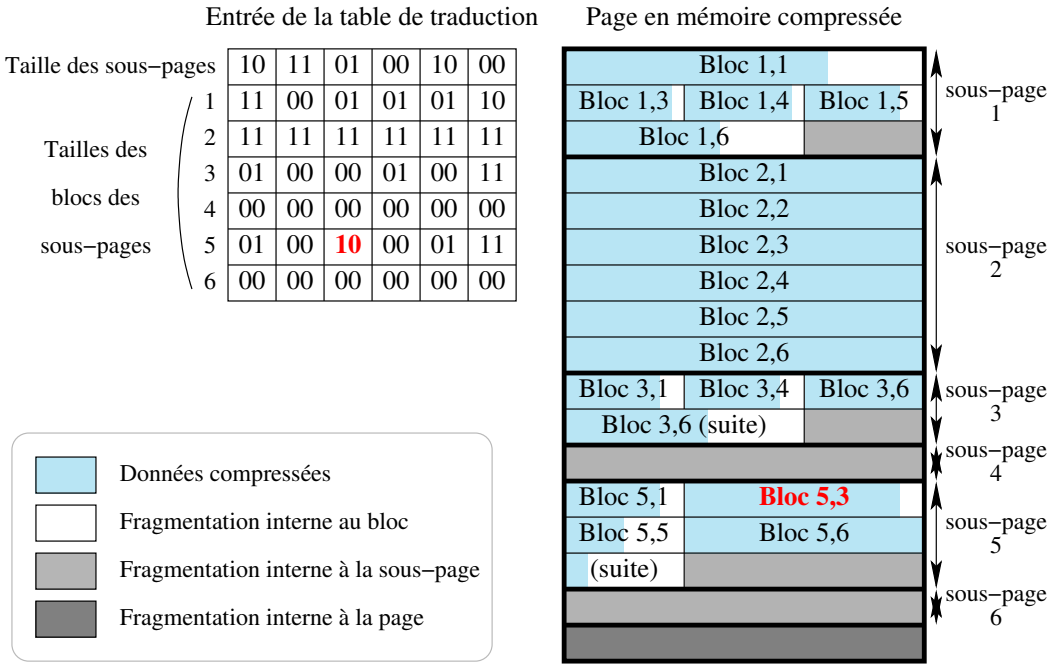


Figure 1.7 – Disposition des sous-pages et des blocs au sein d'une page. 4 tailles de sous-page et quatre tailles de bloc compressé sont possibles. La taille 00 correspondant à une taille nulle, le bloc n'est pas stocké dans la page.

largement inférieur au taux de compression de 50% permis par FPC. Ceci est dû à la fragmentation de l'espace libre. Les performances mesurées en Instructions Par Cycle (IPC) sont plutôt inférieures à celles d'une configuration sans mémoire compressée. L'écart est compris entre $+0,5\%$ (speedup) et $-4,5\%$ (speeddown) avec une moyenne à $-0,5\%$.

Limitations

Cependant, l'architecture proposée n'est pas possible à mettre en œuvre telle que proposée car elle souffre de quatre défauts principaux.

Tout d'abord, la gestion des blocs dont la taille augmente est difficile. De nombreux décalages sont nécessaires. *Ekman et Stenström* proposent de les faire par accès DMA. Cela risque tout de même de saturer la bande passante mémoire.

Ensuite, l'espace libre est fortement fragmenté. Certes, la fragmentation permet souvent de diminuer le nombre de décalages nécessaires lorsque la taille d'un bloc augmente. Mais cela diminue d'autant l'efficacité de la compression.

De plus, l'éviction lors d'une réécriture différée d'un bloc du dernier niveau de cache semble avoir été oubliée. Lorsqu'un bloc est évincé, aucun accès TLB n'est réalisé. Il est donc impossible d'accéder au cache de la table de traduction d'adresse. Il faut donc au moins faire un aller-retour en mémoire pour obtenir la traduction d'adresse.

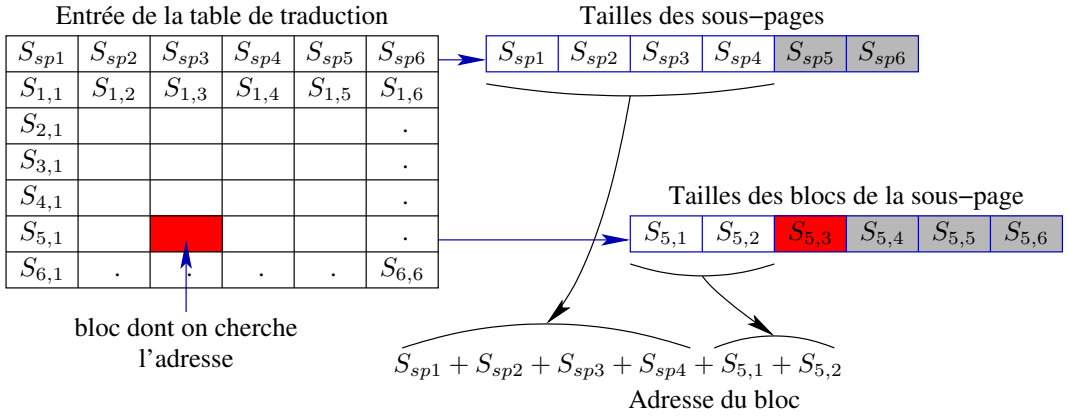


FIGURE 1.8 – Traduction d'adresse. S_{sp_i} est la taille de la sous-page i et $S_{i,j}$ est la taille du je bloc de la ie sous-page. Exemple avec le calcul de l'adresse du 3e bloc de la 5e sous-page obtenu grâce aux tailles des 4 sous-pages et des 2 blocs précédents.

Enfin, cette architecture ne peut pas être utilisée sur un multiprocesseur. Les adresses qui transitent sur le bus sont des adresses en mémoire compressée. Il est impossible d'assurer la cohérence car l'adresse d'un bloc dépend de la taille des blocs le précédant. Pire, les blocs nuls n'ont pas d'adresse. Ils sont stockés directement dans la table de traduction. Pour la même raison, les entrées-sorties ne peuvent pas se faire en DMA.

En conclusion, il ressort de ces observations que la principale modification nécessaire à une mise en œuvre du schéma d'*Ekman et Stenström* permettant la traduction d'adresse lors des éjections du cache et lors des entrées-sorties est d'effectuer la traduction d'adresse dans le contrôleur mémoire et non dans le TLB.

1.4.2.3 Autres propositions

Mesures de *Kjelsø, Gooch et Jones*

En 1996, *Kjelsø, Gooch et Jones* ont mesuré la compressibilité de la mémoire et ont proposé un compresseur matériel permettant d'obtenir un taux de compression de $0,5$ avec une décompression plutôt rapide. Cependant, la compression est appliquée à des pages entières. La latence de décompression est donc critique. Ils ne proposent pas de mécanisme permettant la gestion des pages compressées.

Proposition de *Benini, Bruni, Macii et Macii*

Benini, Bruni, Macii et Macii ont proposé [14, 15] en 2002 puis en 2004 d'utiliser une mémoire compressée afin de réduire la consommation des processeurs embarqués. Une partie de la mémoire est dédiée aux blocs compressés. Si le bloc de taille L une fois

compressé a une taille inférieure à un seuil S, il est stocké dans un emplacement de taille S. Le gain provient de $S \leq L$. Une seule taille de bloc est supportée, ce qui simplifie beaucoup la gestion des blocs compressés.

L'architecture proposée n'augmente pas l'espace disponible pour les applications, mais elle diminue la taille des communications. Les gains sur la consommation sont compris entre 7 et 26%.

Brevet de *Moore*

En 2003, *Moore* a déposé un brevet [66] sur une architecture dans laquelle la mémoire physique est compressée. S'agissant d'un brevet, peu de détails ont été présentés sur le fonctionnement de l'architecture, et aucun résultat n'a été communiqué. Le but de la compression est de diminuer l'utilisation de la bande passante mémoire. Il place la gestion de la mémoire compressée au sein du contrôleur mémoire.

Compression du code

De très nombreuses propositions de compression de la zone de code des applications ont été faites. Elles sont principalement destinées à des architectures embarquées. La compression permet d'atteindre deux objectifs : diminuer le coût de fabrication et augmenter l'autonomie. La compression permet de diminuer la taille et donc le coût de l'EEPROM (ou mémoire flash) dans laquelle est stocké le programme. Elle permet aussi d'économiser de l'énergie. Si le processeur charge moins d'instructions, ou s'il fait moins d'accès à la mémoire principale, le système est plus économe en énergie et l'autonomie s'en trouve augmentée.

Dans certains cas, l'exécutable est complètement décompressé juste avant l'exécution [45, 28, 67, 16, 9, 44, 35]. Dans ce cas, l'objectif consiste seulement à diminuer la taille du code. La décompression peut être faite en logiciel.

Dans d'autres propositions [97, 19, 62, 61, 17, 99], le décodage peut s'effectuer par ligne de cache. Ces architectures permettent en général de diminuer la consommation d'énergie de la hiérarchie mémoire. La décompression est faite dans ce cas par un décompresseur matériel situé entre le cache et la mémoire.

Il existe même des publications [50, 60, 18] proposant d'effectuer le codage instruction par instruction. Dans ce cas, une architecture sans cache peut tout de même utiliser une mémoire compressée.

On peut aussi noter les jeux d'instructions *Thumb* [5] et *Thumb-2* présents sur certains processeurs ARM. Des instructions 16 bits représentant des instructions 32 bits avec des restrictions sur les opérandes sont utilisées pour diminuer la taille du code et l'utilisation de la bande passante mémoire.

La compression de code présente beaucoup moins de contraintes que la compression des données. Le code peut être compressé hors-ligne, par exemple lors de la compilation

ou juste après. Il n'y a donc pas de restriction sur la complexité de l'algorithme de compression qui peut être très lent. Seule la décompression qui est faite à l'exécution doit être rapide. Elle est généralement effectuée par un décompresseur matériel. De plus, aucune écriture n'est réalisée pendant l'exécution. Il n'y a donc pas de changement de taille des blocs compressés à gérer.

1.4.2.4 Formalisation de l'architecture des mémoires compressées

Afin de faire la synthèse des propositions précédemment décrites, nous allons définir l'architecture minimale commune aux mémoires compressées matériellement.

Tout d'abord, il convient de distinguer *deux domaines d'adressage*, que nous nommerons l'espace mémoire physique non-compressé (PNC) et l'espace mémoire physique compressé (PC). Toutes les transactions vues par le processeur ou par les entrées-sorties doivent être dans l'espace mémoire physique non-compressé. Le contrôleur de mémoire compressée assure la conversion des adresses de l'espace physique non-compressé en adresses dans l'espace physique compressé. Nous noterons ces adresses respectivement adresses PNC et adresses PC.

La figure 1.9 page ci-contre illustre ce modèle pour un multiprocesseur à bus partagé. Les processeurs n'ont accès qu'à des adresses PNC. La traduction s'effectue au niveau du contrôleur de mémoire compressée afin d'éviter les problèmes rencontrés par la solution proposée par *Ekman et Stenström*. Ce modèle convient aussi à un système distribué avec des paires d'espaces compressés et non-compressés.

1.4.2.5 Adaptations du système d'exploitation

D'après les propositions précédentes, le système doit être légèrement adapté lors de l'utilisation d'une mémoire compressée.

La compression matérielle de la mémoire ne demande que très peu de support du système d'exploitation. Celui-ci doit seulement être modifié pour éjecter des pages lorsque la mémoire compressée est pleine. Le taux de compression étant variable, la saturation n'arrive pas toujours au même instant. Le système d'exploitation doit donc avoir un compteur, ou doit recevoir un signal lui indiquant la saturation de mémoire.

1.4.3 Machines virtuelles

Récemment des travaux [24, 77, 76] ont montré une forte compressibilité des données dans les machines virtuelles Java. Ce taux de compression élevé a deux origines principales : l'allocation de tableaux et la copie de tableaux.

Lors de la création d'un tableau, une zone mémoire contiguë est implicitement allouée et initialisée à zéro. Toutes les cellules inutilisées du tableau occupent alors de l'espace. *Sartor et al.* ont alors proposé [77, 76] d'utiliser un niveau d'indirection supplémentaire et de découper le tableau *Arraylet* de taille fixe. Un *Arraylet* vide global est

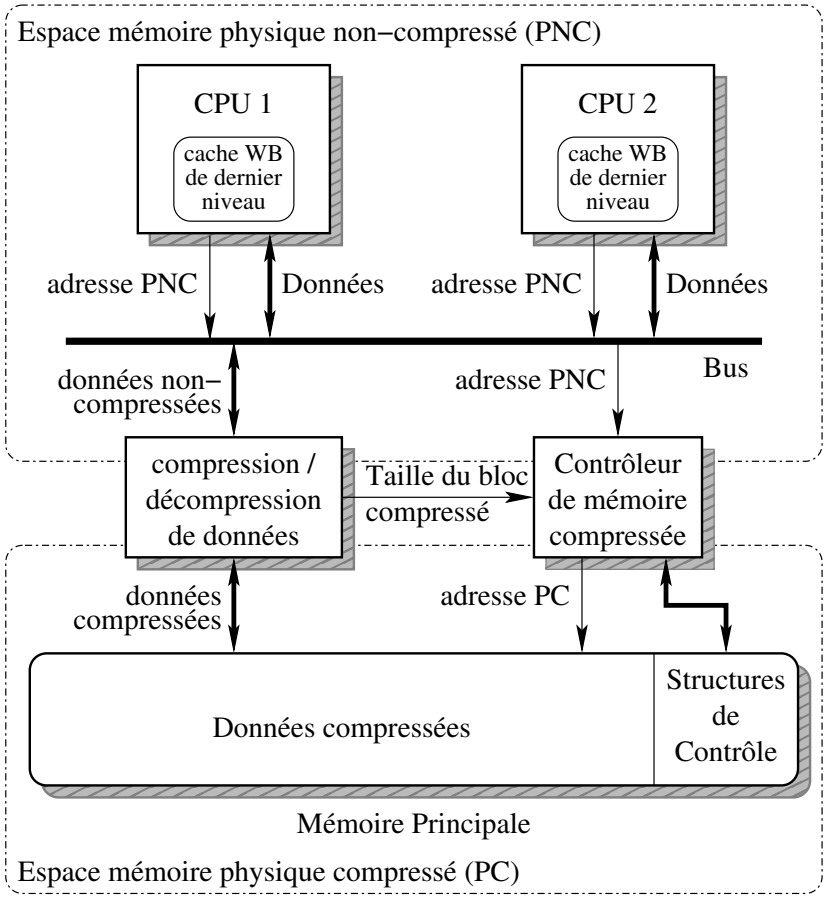


Figure 1.9 – Architecture matérielle des mémoires compressées. Les processeurs ne voient que des adresses dans l'espace non compressé (adresses PNC).

alors défini et partagé par tous les tableaux. Lors d'une écriture réelle par l'application sur l'*Arraylet* vide, celui-ci est recopié ailleurs.

Une fonctionnalité semblable existe déjà en C dans la *glibc* depuis de nombreuses années. La pagination est alors utilisée. Lors d'un appel à *calloc()*, un espace virtuel de la taille de la zone demandée est alloué mais une page physique seulement est réservée [3] et mise à zéro. Toutes les pages virtuelles pointent alors sur cette unique page physique. Lors d'une écriture un mécanisme de Copy-On-Write est utilisé. Grâce à la pagination, l'indirection est faite au niveau de la traduction d'adresse virtuelle en adresse physique, et l'*Arraylet* n'est autre qu'une page.

3. Une seule page physique est allouée, mais le reste de l'espace nécessaire est reservé dans le swap. Cela permet d'éviter la destruction de l'application par l'Out-Of-Memory-Killer lors d'une écriture

La deuxième source de redondance des données identifiée par *Sartor et al.* est la copie de tableau. Java ne prévoit pas de mécanisme de Copy-On-Write sur une partie d'un tableau. Le découpage des tableaux en *Arraylet* permet alors d'effectuer un Copy-On-Write et de ne copier qu'une partie des tableaux. La encore la glibc propose ce mécanisme depuis de nombreuses années grâce à la pagination. Lors d'un appel à *memcpy*, les pages communes à la donnée source et destination ne sont pas dupliquées mais marqués comme Copy-On-Write.

Chapitre 2

Mesure de l'utilisation des blocs nuls

Dans ce document, nous allons proposer des modifications de la hiérarchie mémoire permettant d'exploiter la présence de blocs nuls. Nous avons choisi une taille de bloc identique à la taille d'une ligne de cache, ici 64 octets. Ce choix permet une gestion aisée des blocs nuls puisqu'il n'est pas nécessaire d'utiliser de tampon pour adapter les différentes tailles de bloc manipulées.

Dans ce chapitre, nous commençons par évaluer la proportion de blocs de 64 octets complètement nuls dans les accès aux différents niveaux de caches et à la mémoire. Dans un deuxième temps, nous mesurons l'évolution de ce taux tout au long de l'exécution. Pour finir, nous analyserons pourquoi des applications accèdent à des blocs nuls. Pour cela, nous allons regarder la provenance de ces blocs dans le code source de quelques applications des SPEC CPU 2006.

2.1 Mesure du nombre d'accès à des blocs nuls

A l'aide du simulateur Sesc [73], nous avons mesuré le nombre d'accès à des blocs nuls pour les applications des suites SPEC CPU 2000 et 2006. Ces mesures ont été effectuées lors de l'exécution des cinquante premiers milliards d'instructions. Par souci de concision, seuls les résultats des applications des SPEC CPU 2006 sont détaillés. En effet, nos mesures ont montré que les applications des SPEC CPU 2000 adoptent toutes un comportement que l'on retrouve dans au moins l'une des applications des SPEC CPU 2006. Enfin, l'application *481.wrf* se termine prématurément lors de son exécution dans Sesc, mais les mesures effectuées avec le simulateur Simics au chapitre 4 montrent qu'elle utilise de nombreux blocs nuls et qu'elle a un comportement proche de *459.GemsFDTD*.

Nous avons choisi une durée de simulation suffisamment longue pour que les effets de l'initialisation soient négligeables. Nous le vérifierons dans la section 2.2. La hiérarchie mémoire que nous avons simulée est composée de trois niveaux de caches avec un L1

d'instruction de 64 Ko, un L1 de donnée de 32 Ko, un L2 unifié de 256 Ko et un L3 unifié de 1 Mo. La mémoire principale est choisie suffisamment grande pour contenir toute l'application et ses données.

L'environnement de simulation se base sur Sesc, un simulateur *cycle accurate* MIPS. Il sera décrit plus en détail dans la section 3.2.1 page 60 du chapitre sur le *Zero-Content Augmented Cache*.

	DL1			L2			L3			Mémoire		
Application	ANPKI	APKI	%	ANPKI	APKI	%	ANPKI	APKI	%	ANPKI	APKI	%
400.perlbench	0,1	346	0	0,0	6,4	1	0,0	1,1	4	0,0	0,6	6
401.bzip2	4,0	297	1	0,3	12,9	3	0,3	7,0	4	0,3	3,1	8
403.gcc	96,7	294	33	10,5	20,3	52	6,8	13,7	50	1,3	6,3	20
410.bwaves	101,4	292	35	2,7	13,9	19	1,4	4,6	30	1,4	4,6	30
416.gamess	7,0	301	2	0,3	5,4	6	0,0	0,1	22	0,0	0,0	46
429.mcf	0,3	372	0	0,3	53,0	1	0,3	43,4	1	0,3	39,1	1
433.milc	3,7	404	1	1,6	24,5	7	1,6	24,2	7	1,6	24,2	7
434.zeusmp	108,5	258	42	16,3	22,9	71	5,0	6,9	73	4,4	6,1	73
435.gromacs	2,3	350	1	0,4	8,4	5	0,1	1,6	6	0,1	0,6	9
436.cactusADM	127,2	545	23	3,5	6,4	55	2,9	4,9	59	2,8	4,7	59
437.leslie3d	50,2	280	18	5,4	21,9	25	3,4	15,7	22	2,9	14,2	20
444.namd	0,2	282	0	0,0	8,6	0	0,0	0,2	15	0,0	0,1	23
445.gobmk	10,5	293	4	0,9	9,8	9	0,4	1,9	21	0,2	0,7	24
447.dealII	24,8	306	8	1,6	5,7	27	0,3	2,2	15	0,2	1,2	18
450.soplex	65,4	255	26	9,6	37,2	26	9,3	31,4	30	8,4	24,8	34
453.povray	0,4	349	0	0,0	8,5	0	0,0	0,2	3	0,0	0,0	42
454.calculix	1,9	294	1	0,2	5,6	3	0,1	3,0	4	0,1	0,7	12
456.hmmer	0,1	326	0	0,0	7,3	0	0,0	4,2	0	0,0	3,7	0
458.sjeng	4,2	276	1	0,4	6,8	6	0,4	0,9	44	0,4	0,8	47
459.GemsFDTD	183,9	363	51	23,2	26,7	87	19,7	21,0	94	18,9	19,4	97
462.libquantum	3,7	207	2	0,5	24,8	2	0,5	24,8	2	0,5	24,8	2
464.h264ref	6,8	382	2	0,1	3,4	2	0,0	1,7	2	0,0	1,2	2
465.tonto	6,6	261	3	0,4	8,3	5	0,1	0,9	8	0,0	0,0	16
470.lbm	0,2	253	0	0,1	71,7	0	0,1	40,3	0	0,1	40,3	0
471.omnetpp	17,1	243	7	0,1	31,1	0	0,0	25,9	0	0,0	22,6	0
473.astar	1,2	236	0	0,2	25,9	1	0,2	19,0	1	0,2	6,1	3
482.sphinx3	7,1	245	3	0,7	13,4	5	0,5	10,8	4	0,4	10,1	4
483.xalancbmk	0,3	285	0	0,1	20,8	0	0,0	15,6	0	0,0	12,6	0

Table 2.1 – Accès à des blocs Nuls par Kilo-Instruction (ANPKI) et Accès Par Kilo-Instruction (APKI).

Le tableau 2.1 page ci-contre représente pour les différents niveaux de la hiérarchie mémoire le nombre d'accès à des blocs nuls pour mille instructions (ANPKI), et le nombre d'accès pour mille instructions (APKI). Les accès comprennent non seulement les échecs des niveaux supérieurs mais aussi les *writeback* (réécriture des blocs modifiés). Le cache d'instructions de niveau 1 (IL1) n'est pas représenté car il comporte peu d'accès à des blocs nuls. Cependant, sur les processeurs MIPS [65], l'instruction `nop` [1] est représentée par un mot nul. Les blocs nuls présents proviennent donc de suites d'au moins 16 `nop`. En x86, aucun bloc du cache d'instruction n'est nul. Cependant, une instruction 0x0000 existe [52], il s'agit d'un `add [eax], al` (syntaxe intel, i.e. : `[eax]` ← `[eax] + eax[8:0]`). Son utilité est discutable, et la répétition d'au moins 32 plus qu'improbable. Les instructions `nop` possèdent d'autres codages [2].

Nous observons que la plupart des applications de la suite SPEC CPU 2006 accèdent à des blocs nuls, mais dans des proportions très variables. Pour des applications comme *403.gcc*, *410.bwaves*, *434.zeusmp*, *436.cactusADM*, *450.soplex*, *458.sjeng* et *459.GemsFDTD*, plus de 30% des accès au L3 sont des accès à des blocs nuls. Ces accès existent aussi au niveau du cache de données L1, mais plus l'on descend dans la hiérarchie mémoire, plus ils représentent une part importante des accès. C'est le cas pour l'application *458.sjeng*. La proportion des blocs nuls dans les accès passe de 1 à 47%. Les caches L2 et L3 n'ont pas permis de filtrer ces accès. Cela montre une plus faible localité temporelle des blocs nuls par rapport aux autres blocs.

Certaines applications n'utilisent pratiquement aucun bloc nul. C'est le cas des applications *429.mcf*, *456.hmmer*, *471.omnetpp* et *483.xalancbmk*. Il faut alors en tenir compte, et ne pas considérer d'architectures qui dégradent trop les performances de ces applications.

Le taux de blocs nuls dans les accès est très variable d'une application à l'autre. Pour cette raison, aucune mesure moyenne sur plusieurs applications différentes n'a vraiment de sens. Celle-ci dépendrait plus du jeu d'applications choisi que de l'architecture proposée.

Nous avons effectué ces mêmes mesures avec Simics sur un processeur x86. Malgré les différences importantes entre les architectures x86 et MIPS, comme on pouvait s'y attendre, seul le premier niveau de cache montre des résultats réellement différents. A partir du deuxième niveau de cache, les taux de blocs nuls sont très proches pour la majorité des applications.

1. Représentée par l'instruction `SLL r0, r0, 0`.

2. Sur un processeur x86, hors 64 bits, l'instruction `nop` [53] sur un octet est l'instruction `xchg (e)ax, (e)ax` dont les fausses dépendances en lecture et en écriture sur `ax` sont ignorées. Dans un binaire 64 bits, `xchg eax, eax` est codé différemment car elle doit remettre à zéro les 32 bits de poids fort de `rax`.

2.2 Évolution de l'utilisation de blocs nuls au cours de l'exécution

Afin de nous assurer qu'il s'agit bien d'une utilisation continue des blocs nuls, et non d'un phénomène d'initialisation, nous allons évaluer l'évolution de l'utilisation de blocs nuls au cours de l'exécution. La figure 2.1 page suivante représente, pour les applications des SPEC CPU 2006, le taux de blocs nuls dans les accès au L3 par tranche d'un milliard d'instructions pendant les cinquante premiers milliards d'instructions.

Les applications utilisant moins de 10% de blocs nuls sont représentées sur les graphiques 2.1(a) et 2.1(b). Celles évaluées sur le graphique 2.1(a) n'accèdent à presque aucun bloc nul après la phase d'initialisation. Pour celles du graphique 2.1(b), ce taux varie entre 0 et 10%. Parmi toutes ces applications, celles ayant les plus longues phases d'initialisation sont *462.libquantum*, *464.h264ref* et *473.astar*. Leurs phases d'initialisation durent entre 1 et 2 milliards d'instructions.

Les applications utilisant un peu plus de 10% de blocs nuls sont représentées sur les graphiques 2.1(c) et 2.1(d). Le taux de blocs nuls dans les accès présente des variations tout au long de l'exécution. Ces variations sont particulièrement marquées sur *454.calculix* qui oscille entre 3 et 42%. Pour toutes ces applications, la phase d'initialisation est inférieure à 2 milliards d'instructions.

Trois applications utilisant environ 20% de blocs nuls, ainsi que l'application *403.gcc*, sont représentées sur le graphique 2.1(e). Les trois applications *410.bwaves*, *437.leslie3d* et *445.gobmk* ont un comportement stable tout au long de l'exécution, avec un taux de blocs nuls autour de 20%. Pour toutes ces applications, la phase d'initialisation dure elle aussi moins de 2 milliards d'instructions. L'application *403.gcc* accède quant à elle à de très nombreux blocs nuls, mais de façon extrêmement variable dans le temps : après une initialisation de 2 milliards d'instructions, le taux de blocs nuls oscille entre 5 et 95%.

Les autres applications ayant une large proportion de blocs nuls dans leurs accès sont représentées sur le graphique 2.1(f). Leur taux de blocs nuls est plutôt stable dans le temps. Pour *459.GemsFDTD*, près de 94% des accès au L3 correspondent à des accès à des blocs nuls. Pour *434.zeusmp*, ce taux oscille autour de 75%. Il est d'environ 60% pour *436.cactusADM* et de 30% pour *450.soplex*.

En conclusion, nous constatons une utilisation conséquente des blocs nuls tout au long de l'exécution pour la moitié des applications des SPEC CPU 2006. À l'exception de *403.gcc* et *454.calculix*, on remarque une relative stabilité du taux de blocs nuls dans les accès L3. Toutes les applications simulées ont une période d'initialisation inférieure à 2 milliards d'instructions.

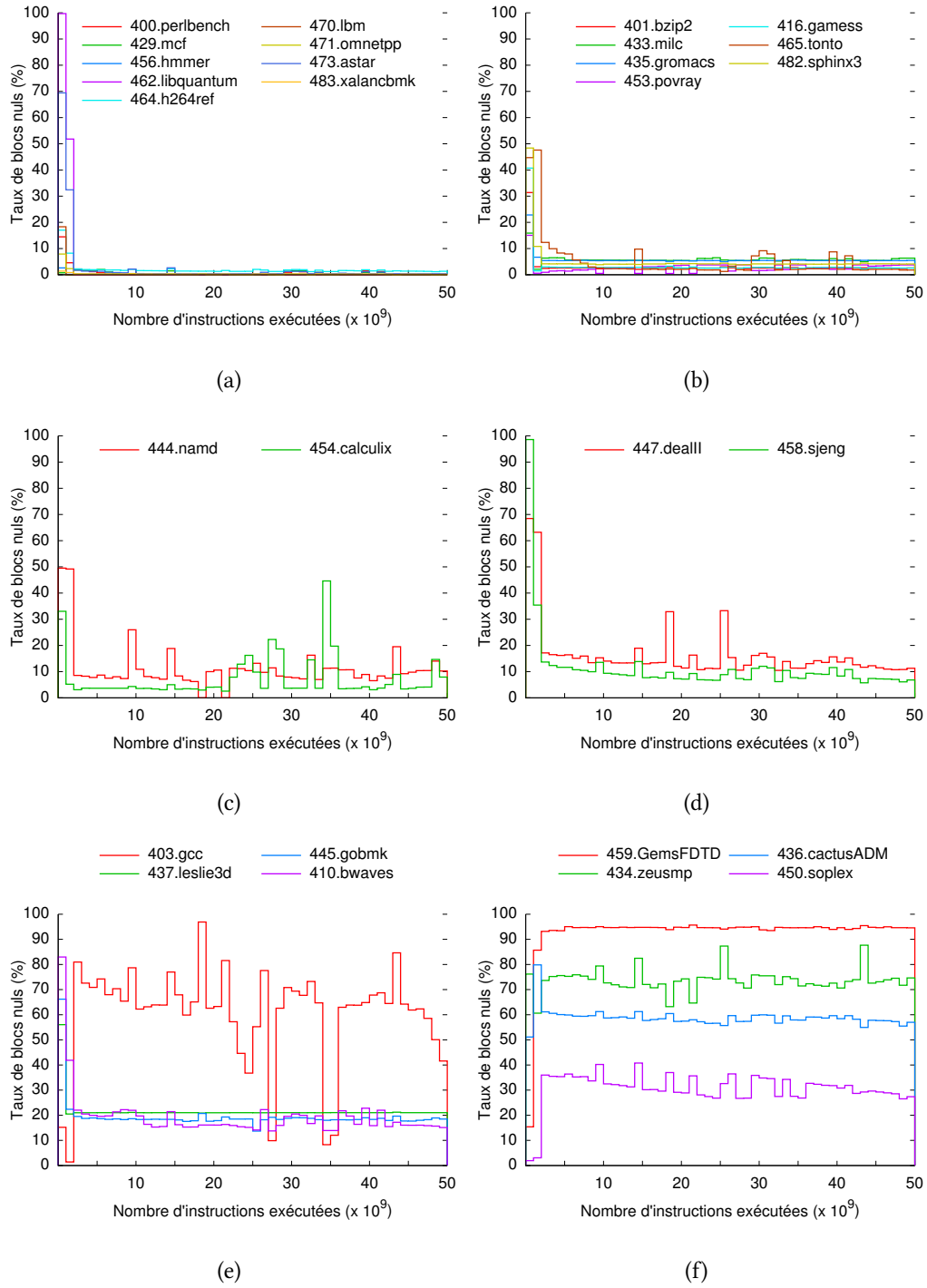


Figure 2.1 – Évolution du taux de blocs nuls dans les accès L3 sur les 50 premiers milliards d'instructions.

2.3 Analyse de la provenance des blocs nuls

Dans cette section, nous chercherons à comprendre pourquoi certaines applications utilisent des blocs nuls tout au long de leur exécution, et non pas seulement à l'initialisation comme il nous a parfois été objecté. Nous allons analyser neuf applications des CPU SPEC 2006 parmi celles qui accèdent à de nombreux blocs nuls : *403.gcc*, *410.bwaves*, *416.gamess*, *434.zeusmp*, *436.cactusADM*, *437.leslie3d*, *450.soplex*, *458.sjeng*, et *459.GemsFDTD*.

Afin de réaliser cette analyse, nous relevons l'adresse des instructions accédant à des blocs nuls à chaque niveau de cache. Pour les *writebacks*, c'est l'adresse de la dernière écriture dans la ligne de cache qui est prise en compte.

Pour une partie de ces applications, nous présentons un tableau avec le nom des principales fonctions accédant à des blocs nuls et les adresses des instructions *Load* ou *Store*. De plus, pour chaque niveau de cache et chacune de ces instructions, nous présenterons le taux d'accès à un bloc nul (T_n), la proportion des accès nuls faits par l'instruction (P_{an}) et la proportion des accès faits par l'instruction (P_a). Ces taux sont définis par :

$$Taux\ d'accès\ à\ un\ bloc\ nul : T_n = \frac{nb\ accès\ à\ un\ bloc\ nul}{nb\ exécutions\ de\ l'instruction}$$

$$Proportion\ des\ accès\ nuls : P_{an} = \frac{nb\ accès\ à\ un\ bloc\ nul}{nb\ accès\ nuls\ total\ au\ cache}$$

$$Proportion\ des\ accès : P_a = \frac{accès\ à\ un\ bloc\ nul}{nb\ accès\ total\ au\ cache}$$

2.3.1 403.gcc

403.gcc est une application écrite en C effectuant des calculs sur des entiers. Il s'agit du compilateur *gcc* dans sa version 3.2. Il utilise de nombreux blocs nuls tout au long de son exécution.

Les principales fonctions accédant à des blocs nuls sont représentées dans le tableau 2.2 page ci-contre. Ce sont les fonctions *memset*, *cselib_init*, *sbitmap_not*, *sbitmap_union_of_diff*, *loop_regs_scan* et *scan_loop*.

La fonction *memset* est la principale fonction accédant à des blocs nuls. A elle seule, elle représente 35% des accès au L3, et 69% des accès L3 à des blocs nuls. *memset* est appelée en 205 sites dans le code, dont 203 avec la valeur `0` et deux avec la valeur `-1`. Elle écrit des blocs nuls au travers de deux instructions `store`. Tout au long de l'exécution, *memset* est utilisée pour initialiser des structures temporaires : il s'agit principalement des nœuds dans l'arbre de représentation interne de *gcc* et des nombreuses structures représentant les registres libres ou réservés lors de l'allocation de registres. Les appels à

Fonction	Adresse Ld/St	DL1			L2			L3			Mémoire		
		T_n	P_{an}	P_a	T_n	P_{an}	P_a	T_n	P_{an}	P_a	T_n	P_{an}	P_a
memset	0x0077692c	98	31	10	98	35	18	97	42	21	81	26	5
	0x00776924	98	31	10	98	35	18	97	27	14	83	25	5
cselib_init	0x004b0030	100	20	7	100	23	12	100	21	11	100	0	0
sbitmap_not	0x006c8b0c	88	1	0	79	0	0	79	1	0	79	3	1
sbitmap_union_of_diff	0x006c8b00	84	1	0	76	0	0	76	1	0	76	3	1
	0x006c8b50	10	0	0	77	0	0	77	1	0	77	3	1
loop_regs_scan	0x00635888	100	0	0	100	1	0	100	1	0	100	5	1
	0x00635880	100	0	0	100	1	0	100	1	0	100	5	1
scan_loop	0x0063ec28	100	0	0	100	1	0	100	1	0	100	5	1

Table 2.2 – Localisation des accès nuls dans 403.gcc

memset aux différentes phases de l'exécution expliquent l'irrégularité de la courbe 2.1(e) page 41.

La deuxième fonction accédant à de nombreux blocs nuls est *cselib_init*. Elle réinitialise des variables internes lors des passes de recherche de sous-expressions communes. Cette fonction est appelée pour chaque bloc de base. Elle accède à des structures tenant dans le cache de niveau 3 de 1 Mo, ce qui explique qu'elle représente 20% des accès nuls du L1 mais moins de 1% des accès nuls à la mémoire (tableau 2.2).

Les fonctions sbitmap_* manipulent des champs de bits. Ces derniers ont de multiples utilisations, notamment lors du parcours de l'arbre de représentation interne pour marquer les nœuds visités, lors de l'ordonnancement des instructions dans un bloc de base, etc. Ces champs de bits sont fréquemment remis à zéro.

2.3.2 410.bwaves

410.bwaves est une application écrite en Fortran 77 travaillant flottants de 64 bits. Elle simule numériquement la propagation d'ondes de chocs dans un milieu à écoulement visqueux. Avec le jeu d'entrées *ref*, l'espace est modélisé par un maillage de $65 \times 65 \times 64$ cubes. La plupart des cubes sont initialisés à la valeur décimale `0.0d0`. Or, la représentation en mémoire de la valeur décimale `0.0d0` est la même que celle de la valeur entière `0`, à savoir 64 bits mis à `0`. Cette application se termine après avoir effectué les calculs pour cent incréments de temps.

Dans le tableau 2.3 page suivante, la principale fonction accédant à des blocs nuls est *mat_times_vec*. Comme son nom l'indique, elle effectue le produit d'une matrice par un vecteur. A elle seule, elle représente plus de 16% des accès nuls au L3. Cela s'explique par le fait qu'elle soit fréquemment appelée par la fonction *bi_cgstab_block* avec en paramètre un vecteur nul.

La fonction *bi_cgstab_block*, dont un extrait de code est donné figure 2.2 page 45, est appelée à chaque fois que le temps est incrémenté, c'est-à-dire cent fois pour une

Fonction	Adresse Ld/St	DL1			L2			L3			Mémoire		
		T_n	P_{an}	P_a	T_n	P_{an}	P_a	T_n	P_{an}	P_a	T_n	P_{an}	P_a
mat_times_vec	0x004007e4	77	0	0	90	2	0	91	4	1	91	4	1
	0x00400814	75	2	1	75	2	0	75	3	1	75	3	1
	0x0040081c	75	2	1	75	2	0	75	3	1	75	3	1
	0x00400838	75	2	1	74	2	0	74	3	1	74	3	1
	0x0040087c	75	0	0	73	2	0	73	3	1	73	3	1
bi_cgstab_block	0x004015c0	87	0	0	91	1	0	91	2	0	91	2	0
	0x00401324	90	0	0	90	1	0	90	2	0	90	2	0
	0x004014a0	90	0	0	90	1	0	90	2	0	90	2	0
	0x004015bc	90	0	0	90	1	0	90	2	0	90	2	0
	0x00401950	86	0	0	86	1	0	86	1	0	86	1	0
	0x00401328	86	0	0	86	1	0	86	1	0	86	1	0
	0x004015c4	86	0	0	86	1	0	86	1	0	86	1	0
	0x004015e0	86	0	0	86	1	0	86	1	0	86	1	0
	0x004016cc	85	0	0	86	1	0	86	1	0	86	1	0
	0x00401958	83	0	0	84	1	0	84	1	0	84	1	0
	0x004014a4	84	0	0	84	1	0	84	1	0	84	1	0
	0x004016c8	84	0	0	84	1	0	84	1	0	84	1	0
	0x004016e0	85	0	0	84	1	0	84	1	0	84	1	0
	0x0040183c	84	0	0	84	1	0	84	1	0	84	1	0
	0x00401838	83	0	0	83	1	0	83	1	0	83	1	0
	0x00401978	83	0	0	83	1	0	83	1	0	83	1	0
	0x00401a64	83	0	0	83	1	0	83	1	0	83	1	0
	0x00401954	83	0	0	83	1	0	83	1	0	83	1	0
	0x004010ac	91	0	0	92	0	0	92	1	0	92	1	0

Table 2.3 – Localisation des accès nuls dans 410.bwaves

exécution complète de l'application. Une partie des `store` accédant à des blocs nuls est représentée dans le tableau 2.3. Ces écritures ne sont pas filtrées par les caches, ainsi le nombre d'accès est constant quelque soit le niveau. Dans l'extrait de code présenté, on remarque que la fonction *bi_cgstab_block* réinitialise de nombreux vecteurs (aux lignes 5, 20, 21, et 22). Pour le jeu d'entrées *ref*, chacun de ces vecteurs occupe $65*65*64*5*8 = 10816000$ octets, soit environ $10,3$ Mo. On remarque aussi à la ligne 11 l'appel à la fonction *mat_times_vec* avec en paramètre le vecteur r tout juste réinitialisé à `0`.

2.3.3 416.gamess

L'application *416.gamess* est écrite en Fortran travaillant sur des flottants 64 bits. Elle réalise des calculs de chimie quantique sur de larges matrices.

D'après le tableau 2.4 page 46, la principale fonction accédant à des blocs nuls est *atoms*. Celle-ci est appelée régulièrement au cours de l'exécution. Les vingt-deux instruc-

```
do k=1,nz !dimension z de la grille (= 64 pour le jeu d'entrées 'ref')
   do j=1,ny !dimension y de la grille (= 65 pour 'ref')
      do i=1,nx !dimension x de la grille (= 65 pour 'ref')
         do l=1,nb !nb constante initialisée à 5
            r(l,i,j,k)=0.0d0
         enddo
      enddo
   enddo
enddo

call mat_times_vec(r,x,a,axp,ayp,azp,axm,aym,azm,nb,nx,ny,nz)
r2=0.0d0

do k=1,nz !dimension z de la grille (= 64 pour le jeu d'entrées 'ref')
   do j=1,ny !dimension y de la grille (= 65 pour 'ref')
      do i=1,nx !dimension x de la grille (= 65 pour 'ref')
         do l=1,nb !nb constante initialisée à 5
            r(l,i,j,k) = b(l,i,j,k) - r(l,i,j,k)
            r2 =r2+r(l,i,j,k)**2
            p(l,i,j,k) = 0.0d0
            v(l,i,j,k) = 0.0d0
            t(l,i,j,k)=0.0d0
            rhat(l,i,j,k) = r(l,i,j,k)
         enddo
      enddo
   enddo
enddo
```

Figure 2.2 – 410.bwaves : Extrait de code de la fonction *bi_cgstab_block*.

tions comprises entre `0x0073ad40` et `0x0073ad94` sont toutes des écritures de blocs nuls. Le code correspondant est donné figure 2.3 page suivante. Il effectue la remise à zéro de onze tableaux occupant chacun 40 Ko. Ces instructions représentent plus de 20% des accès nuls à la mémoire. On peut noter que quatre instructions `store` ne provoquent pas d'accès à la mémoire. Dans le L3, elles se sont toutes retrouvées masquées par une écriture suivante sur la même ligne.

La deuxième fonction accédant à de très nombreux blocs nuls est *vclr*. Cette fonction, dont le nom signifie *vector clear*, remet à zéro un vecteur. Elle possède 990 sites d'appel dans le code.

Fonction	Adresse Ld/St	DL1			L2			L3			Mémoire		
		T_n	P_{an}	P_a	T_n	P_{an}	P_a	T_n	P_{an}	P_a	T_n	P_{an}	P_a
atoms	0x0073ad40	100	0	0	100	0	0	100	0	0	100	1	1
	0x0073ad44	100	0	0	100	0	0	100	0	0	-	0	0
	0x0073ad48	100	0	0	100	0	0	100	0	0	100	1	1
	0x0073ad4c	100	0	0	100	0	0	100	0	0	-	0	0
	0x0073ad50	100	0	0	100	0	0	100	0	0	100	1	1
	0x0073ad54	100	0	0	100	0	0	100	0	0	100	0	0
	0x0073ad58	100	0	0	100	0	0	100	0	0	100	1	1
	0x0073ad5c	100	0	0	100	0	0	100	0	0	-	0	0
	0x0073ad60	100	0	0	100	0	0	100	0	0	100	1	1
	0x0073ad64	100	0	0	100	0	0	100	0	0	100	1	1
	0x0073ad68	100	0	0	100	0	0	100	0	0	100	1	1
	0x0073ad6c	100	0	0	100	0	0	100	0	0	100	1	1
	0x0073ad70	100	0	0	100	0	0	100	0	0	100	1	1
	0x0073ad74	100	0	0	100	0	0	100	0	0	100	1	1
	0x0073ad78	100	0	0	100	0	0	100	0	0	100	1	1
	0x0073ad7c	100	0	0	100	0	0	100	0	0	100	1	1
	0x0073ad80	100	0	0	100	0	0	100	0	0	100	1	1
	0x0073ad84	100	0	0	100	0	0	100	0	0	100	1	1
	0x0073ad88	100	0	0	100	0	0	100	0	0	100	1	1
	0x0073ad8c	100	0	0	100	0	0	100	0	0	100	1	1
	0x0073ad90	100	0	0	100	0	0	100	0	0	100	1	1
	0x0073ad94	100	0	0	100	0	0	100	0	0	-	0	0
vclr	0x008f18a4	42	3	0	40	4	0	36	2	0	97	15	7

Table 2.4 – Localisation des accès nuls dans 416.gamess

```
      !EX, CS, CP, CD, CF, CG, CSINP, CPINP, CDINP, CFINP, CGINP
      !tableaux de 5000 flottants 64bits
       IUNT = IUNTRD
       DO 100 I = 1,MXGTOT !début boucle: 5000 itérations
          EX(I) = ZERO
          CS(I) = ZERO
          CP(I) = ZERO
          CD(I) = ZERO
          CF(I) = ZERO
          CG(I) = ZERO
          CSINP(I) = ZERO
          CPINP(I) = ZERO
          CDINP(I) = ZERO
          CFINP(I) = ZERO
          CGINP(I) = ZERO
  100 CONTINUE !fin de boucle
```

Figure 2.3 – 416.gamess : Extrait de code de la fonction *atoms*.

2.3.4 434.zeusmp

434.zeusmp est une application écrite en Fortran 77 travaillant sur des flottants de 64 bits. Il s'agit d'un simulateur de mécanique des fluides utilisé dans la simulation de phénomènes astrophysiques.

Pour cette application, aucune fonction ne se dégage des autres par son utilisation de blocs nuls. Il n'y a pas de réinitialisation périodique de variables comme dans *410.bwaves* et *416.gamess*. Les accès à des blocs nuls proviennent de valeurs nulles présentes dans l'espace modélisé qui restent nulles durant l'exécution.

2.3.5 436.cactus

436.cactusADM est une application écrite en C et en Fortran 90 travaillant sur des flottants de 64 bits. C'est une combinaison de Cactus, un environnement de résolution de problèmes, et de BenchADM, un noyau de calcul de nombreuses applications de la théorie de la relativité. CactusADM résout les équations d'évolution d'Einstein qui décrivent comment l'espace-temps se courbe en réponse à sa teneur en matière.

Comme pour *434.zeusmp*, peu de fonctions se démarquent dans cette application. *Bench_StaggeredLeapfrog2* est celle qui accède au plus grand nombre de blocs nuls. Cette fonction est d'une complexité extrême. Un de ses *basic blocks* compte 3224 instructions (dont 1499 instructions `load` et 830 instructions `store`). Il n'existe pas d'instruction `store` n'écrivant que des zéros. Comme *434.zeusmp*, cette application ne réinitialise pas ses variables, les blocs nuls proviennent donc de données qui restent nulles. Cette application utilise intensivement la hiérarchie mémoire. En effet, le tableau 2.1 page 41 montre que 54, 5% des instructions sont des accès mémoire.

2.3.6 437.leslie3d

437.leslie3d est une application écrite en Fortran 90 travaillant sur des flottants 64 bits. C'est un simulateur de mécanique des fluides. Il est utilisé pour modéliser le mélange de fluides lors de l'injection dans une chambre de combustion.

Pour cette application aussi, aucune fonction ne se démarque. Elle n'effectue pas de réinitialisation périodique. Toutes les fonctions effectuant des traitements accèdent régulièrement à des blocs nuls.

2.3.7 450.soplex

450.soplex est un solveur linéaire écrit en C++ utilisant l'algorithme du simplexe. Les données d'entrées sont représentées par une matrice creuse de flottants double-précision.

La fonction *memset* est la fonction accédant le plus souvent aux blocs nuls. Elle représente entre 10 et 12% des accès à chaque niveau de cache.

Fonction	Adresse Ld/St	DL1 T_n	DL1 P_{an}	DL1 P_a	L2 T_n	L2 P_{an}	L2 P_a	L3 T_n	L3 P_{an}	L3 P_a	Mémoire T_n	Mémoire P_{an}	Mémoire P_a
memset	0x00511414	98	38	10	97	32	8	97	33	10	97	37	12
	0x0051141c	98	38	10	100	33	9	100	34	10	100	36	12
assign2prod…	0x00467050	60	1	0	72	5	1	88	6	2	99	6	2
selectLeave	0x0045bf04	22	3	1	22	2	1	22	2	1	22	2	1

Table 2.5 – Localisation des accès nuls dans 450.soplex

```
1    /// set vector to 0.
2    void clear()
3    {
4      if (dimen)
5         memset(val, 0, dimen*sizeof(Real));
6    }
```

Figure 2.4 – 450.soplex : Extrait de code de la classe *Vector*.

Cette fonction n'est appelée que dans la méthode *Vector : :clear()* dont le code est donné figure 2.4. Il y a quatre-vingts sites d'appels pour cette méthode. *450.soplex* appelle régulièrement *Vector : :clear()* afin de réinitialiser ses variables temporaires tout au long de l'exécution.

Parmi les autres fonctions accédant à des blocs nuls, *SSVector : :assign2product4setup* et *SPxSteepPR : :selectLeave* sont souvent utilisées juste après un appel à *Vector : :clear()*. De ce fait, certains de leurs paramètres sont des vecteurs nuls.

2.3.8 458.sjeng

458.sjeng est une application écrite en C travaillant sur des entiers. Elle joue aux échecs en évaluant pour un jeu donné les différents mouvements possibles et leurs implications sur plusieurs coups.

Fonction	Adresse Ld/St	DL1 T_n	DL1 P_{an}	DL1 P_a	L2 T_n	L2 P_{an}	L2 P_a	L3 T_n	L3 P_{an}	L3 P_a	Mémoire T_n	Mémoire P_{an}	Mémoire P_a
memset	0x00437d5c	64	33	0	98	41	2	100	42	18	100	42	20
memset	0x00437d54	63	33	0	76	41	2	100	42	18	100	42	20
ProbeTT	0x0041e6ac	84	1	0	85	6	0	86	6	3	86	6	3
ProbeTT	0x0041e694	34	0	0	35	3	0	37	3	2	38	3	2

Table 2.6 – Localisation des accès nuls dans 458.sjeng

La fonction *memset* représente 66 à 84% des accès à des blocs nuls grâce à deux instructions `store`. Cette application réinitialise ses variables à chaque évaluation d'un coup. Dans le tableau 2.6, on peut noter que P_a augmente lorsque l'on descend dans la hiérarchie mémoire. Cela traduit un très fort taux d'échec sur les accès aux caches de ces deux instructions `store`.

2.3.9 459.GemsFDTD

459.GemsFDTD est une application écrite en Fortran 90 travaillant sur des flottants de 64 bits. Elle résout les équations de Maxwell en 3D dans le domaine temporel. Elle n'effectue pas de remise à zéro périodique. Les fonctions de la boucle de calcul principale accèdent à des matrices dont les valeurs sont presque toutes nulles.

2.4 Synthèse

Application	Langage	Type
400.perlbench	C	Entiers
401.bzip2	C	Entiers
429.mcf	C	Entiers
433.milc	C	Flottants
435.gromacs	C/Fortran	Flottants
453.povray	C++	Flottants
454.calculix	C/Fortran	Flottants
456.hmmer	C	Entiers

Application	Langage	Type
462.libquantum	C	Entiers
464.h264ref	C	Entiers
465.tonto	Fortran	Flottants
470.lbm	C	Flottants
471.omnetpp	C++	Entiers
473.astar	C++	Entiers
482.sphinx3	C	Flottants
483.xalancbmk	C++	Entiers

Table 2.7 – Applications accédant à moins de 10% de blocs nuls

Application	Langage	Type	Taux Nul	Réinit.
403.gcc	C	Entiers	50 %	Oui
410.bwaves	Fortran	Flottants	30 %	Oui
416.gamess	Fortran	Flottants	22 %	Oui
434.zeusmp	Fortran	Flottants	73 %	Non
436.cactusADM	C/Fortran	Flottants	59 %	Non
437.leslie3d	Fortran	Flottants	22 %	Non
444.namd	C++	Flottants	15 %	Non
445.gobmk	C	Entiers	21 %	Oui
447.dealII	C++	Flottants	15 %	Oui
450.soplex	C++	Flottants	30 %	Oui
458.sjeng	C	Entiers	44 %	Oui
459.GemsFDTD	Fortran	Flottants	94 %	Non

Table 2.8 – Applications accédant de nombreux blocs nuls

Les tableaux 2.7 et 2.8 présentent une synthèse des observations pour toutes les applications de la suite SPEC CPU 2006.

Les applications représentées dans le tableau 2.8 accèdent à de nombreux blocs nuls tout au long de leur exécution. Il s'agit d'applications travaillant aussi bien sur des entiers

que sur des flottants. Ceci est permis grâce à la représentation du `0` des flottants par un mot nul. En effet, par défaut, `0` vaut `+0`, le `-0` étant codé différemment (`0x80000000`). Le langage de programmation n'a pas non plus d'impact particulier : certaines applications sont en C, C++ et d'autres Fortran.

On peut cependant distinguer deux comportements d'applications : celles qui réinitialisent des variables temporaires fréquemment et celles qui effectuent des calculs sur des jeux de donnés majoritairement nuls. Ces dernières ont un taux de blocs nuls plus homogène lors de leur exécution.

En conclusion, dans ce chapitre nous avons montré que l'accès à un nombre conséquent de blocs nuls dans certaines applications n'est pas un simple phénomène d'initialisation, mais que des blocs nuls sont présents tout au long de l'exécution de certaines applications. Nous avons mesuré une période d'initialisation inférieure à deux milliards d'instructions pour toutes les applications. Passée cette période, le taux de blocs nuls reste important pour plus de la moitié des applications de la suite SPEC CPU 2006. Une gestion efficace de ces blocs nuls peut donc se révéler intéressante en terme de performances et de réduction de bande passante mémoire.

Chapitre 3

Zero-Content Augmented Cache

La hiérarchie mémoire tient une place centrale dans le fonctionnement d'un système. Son rôle sur les performances globales devient de plus en plus importante. Depuis les années 1980, l'augmentation de la fréquence des processeurs a créé un véritable écart entre le nombre de cycles nécessaires pour accéder à une donnée présente dans le cache et à une donnée stockée dans la mémoire principale. Cette latence mémoire importante a généralisé l'utilisation des caches au milieu des années 1980, puis l'exécution dans le désordre au milieu des années 1990. Les caches ont alors pour principal but de diminuer la latence d'accès. Au milieu des années 2000, devant la difficulté d'augmenter la fréquence, les architectures multicœurs se sont généralisées. Depuis, le nombre de cœurs d'exécution gérant chacun plusieurs processus ne cesse d'augmenter. Les différents processus s'exécutant simultanément partagent une partie de la hiérarchie mémoire, augmentant sérieusement la pression sur celle-ci. Les caches se voient alors confier le rôle de filtrer le nombre d'accès, afin d'éviter la saturation de la bande passante des derniers niveaux de la hiérarchie mémoire. L'efficacité des derniers niveaux de cache est donc un réel enjeu.

Dans ce chapitre, nous proposons une modification de l'architecture mémoire permettant d'exploiter la part importante de blocs nuls présents dans les accès mémoire. Nous avons constaté dans le chapitre précédent que pour environ la moitié des applications une part importante des accès sont des accès à des blocs de 64 octets complètement nuls. Mémoriser des blocs nuls dans la hiérarchie mémoire peut être considéré comme un gaspillage de ressources.

Dans notre publication [31], nous avons proposé l'utilisation d'un petit cache ne stockant que les blocs nuls, appelé cache de zéros. Nous avons proposé d'utiliser ce cache de zéros conjointement avec le dernier niveau de cache, au sein d'un *Zero-Content Augmented Cache*. Ce cache permet d'augmenter les performances (mesurées en instructions par cycle) et de réduire le nombre d'accès mémoire pour les applications accédant à de nombreux blocs nuls, mais ne pénalise pas les applications n'accédant qu'à très peu de blocs nuls.

Ce chapitre est découpé en trois parties. Dans un premier temps, nous allons présenter l'architecture du *Zero-Content Augmented Cache*. Dans un deuxième temps nous allons faire une analyse détaillée des performances de notre proposition. Enfin, dans une troisième partie, nous allons proposer de fusionner le cache de zéros au sein du cache principal afin de quasiment supprimer l'intégralité du coût matériel de notre proposition.

3.1 Architecture du Zero-Content Augmented Cache

Dans le chapitre 2, nous avons constaté que pour certaines applications une part significative des accès aux caches et à la mémoire sont des accès à des blocs nuls. Ce taux augmente lorsque l'on descend dans la hiérarchie mémoire. Nous verrons dans la section 3.2.3.4 que ces blocs nuls ont une forte localité spatiale.

Dans cette section, nous allons présenter l'architecture du *Zero-Content Augmented Cache* qui tire parti des blocs nuls pour libérer de l'espace dans le cache et augmenter le taux de *hits* sur ces blocs.

3.1.1 Structure

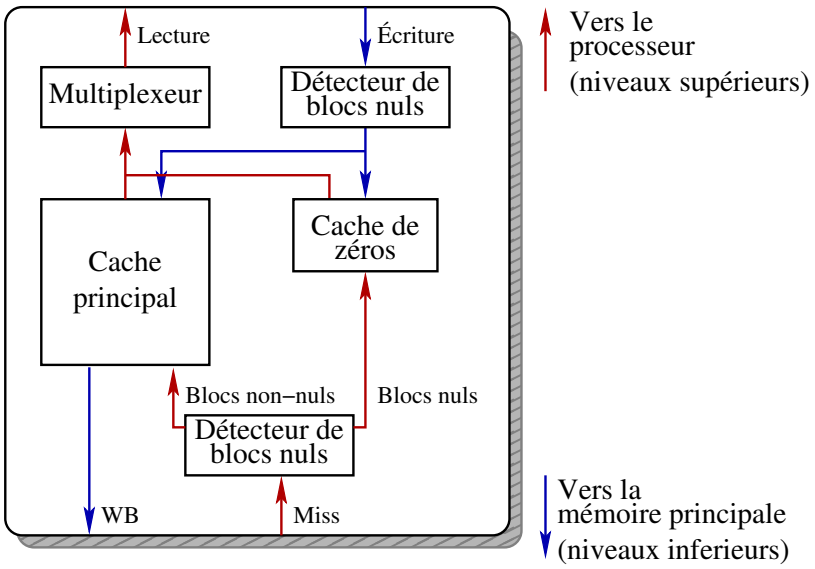


Figure 3.1 – Structure du Zero-Content Augmented Cache

Un *Zero-Content Augmented Cache* est représenté figure 3.1. Il est constitué d'un cache principal, d'un cache de zéros et de détecteurs de blocs nuls.

Le cache principal est un cache traditionnel. Aucune modification ne lui est apportée. Il occupe la majeure partie de la surface du *Zero-Content Augmented Cache*.

Le cache de zéros est un cache spécialisé destiné à stocker les blocs nuls. Afin de minimiser le coût relatif du *tag* par rapport au simple bit nécessaire pour représenter un bloc nul, ce cache a une structure de cache à secteurs. Nous noterons N le nombre de blocs par secteur. Chaque entrée est composée d'un *tag* d'adresse correspondant au secteur de N-blocs et de N bits de validité, c'est-à-dire un bit de validité par bloc du secteur. Un bloc n'est présent dans ce cache que si son bit de validité est vrai et que le secteur le contenant est présent.

Les détecteurs de blocs nuls effectuent des tests de nullités. Ces test sont effectués lors des écritures de blocs provenant du processeur (ou des niveaux supérieurs), ainsi que lors de l'arrivée d'un bloc de la mémoire (ou des niveaux inférieurs), si un défaut de cache a eu lieu. Ce détecteur est très simple et rapide : il ne nécessite que l'application d'un OU global sur la valeur du bloc.

Nous allons maintenant présenter le fonctionnement du *Zero-Content Augmented Cache*, ce qui nous conduira à aborder dans un premier temps les lectures, puis ensuite les écritures.

3.1.2 Accès en lecture

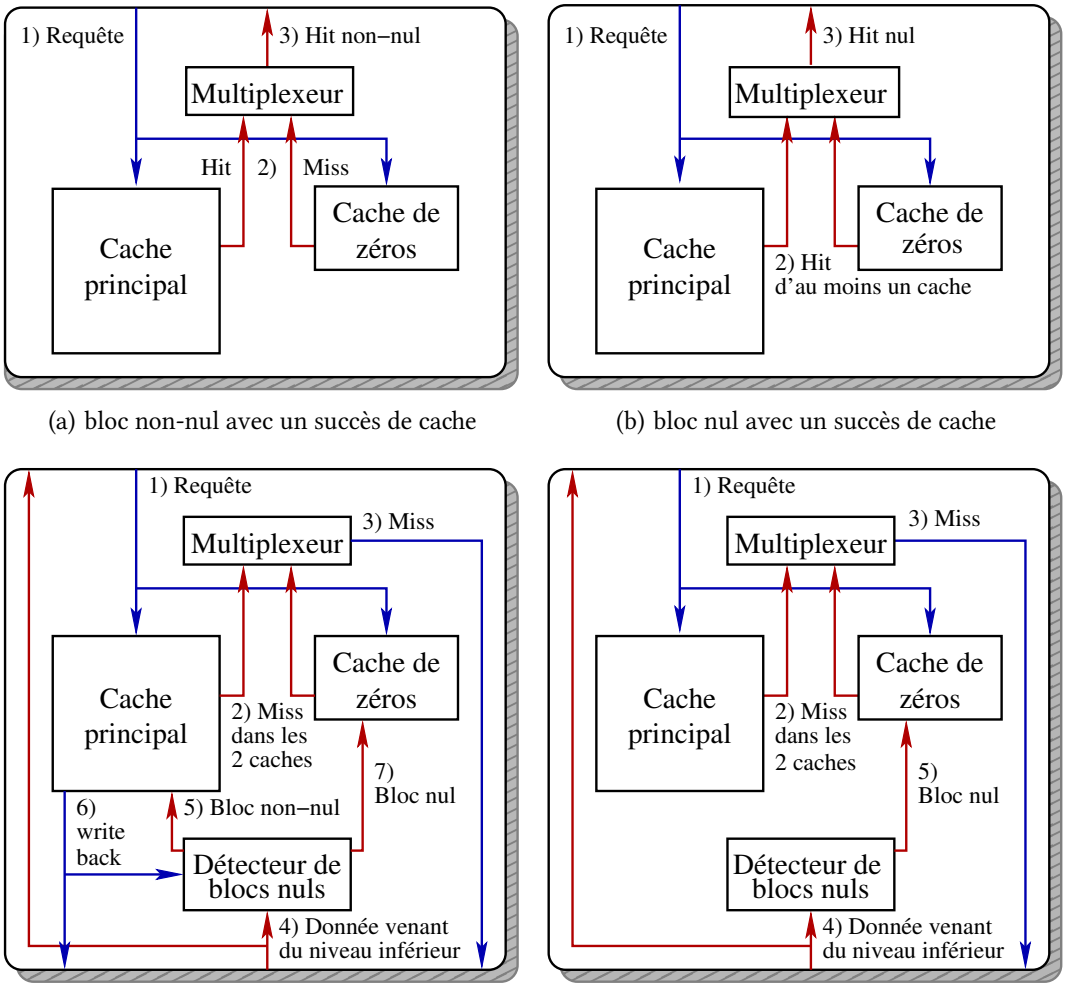
Lors d'une lecture dans le *Zero-Content Augmented Cache*, le cache principal et le cache de zéros sont interrogés simultanément. Il suffit que l'un des deux contienne la donnée pour obtenir un succès de cache. Le cache de zéros ne peut contenir que des blocs nuls, alors que le cache principal peut contenir aussi bien des blocs nuls que non-nuls.

On peut distinguer six cas selon que le bloc est nul ou non-nul et selon les échecs ou succès des deux caches. Ces différents cas sont représentés dans le tableau 3.1. Les trois cas d'une ligne nulle provoquant un succès dans l'un des deux caches sont regroupés ensemble sur la figure 3.2 page suivante.

Bloc	Cache Principal	Cache de Zéros	Figure 3.2
non-nul	Succès	Échec	3.2(a)
non-nul	Échec	Échec	3.2(c)
nul	Succès	Échec	3.2(b)
nul	Échec	Succès	3.2(b)
nul	Succès	Succès	3.2(b)
nul	Échec	Échec	3.2(d)

Table 3.1 – Différents cas possibles selon la nullité du bloc et les succès/échecs du cache principal et du cache de zéros.

Les accès en lecture sont représentés sur la figure 3.2 page suivante :



(a) bloc non-nul avec un succès de cache

(b) bloc nul avec un succès de cache

(c) bloc non-nul avec un défaut de cache

(d) bloc nul avec un défaut de cache

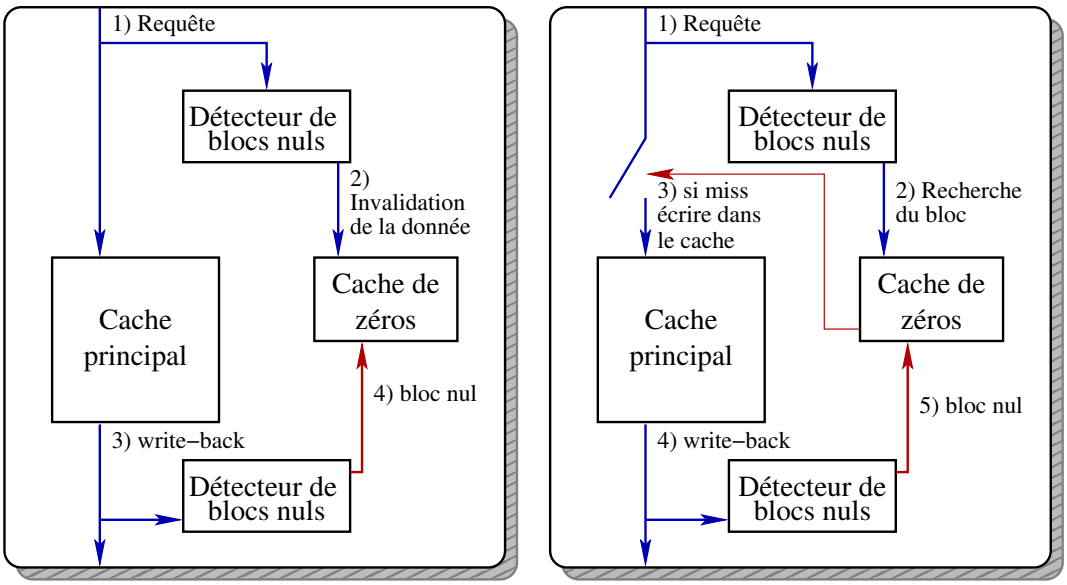
Figure 3.2 – Lecture d'un bloc nul/non-nul avec succès/défaut dans le ZCA Cache

- Lors de la lecture d'un bloc non-nul, seul le cache principal peut contenir le bloc demandé. En cas de succès, le bloc est retourné, figure 3.2(a). En cas d'échec, figure 3.2(c), le bloc est demandé au niveau inférieur (3) ; lors de son retour (4), il passe dans le détecteur de blocs nuls ; comme il est non-nul, il est donc stocké dans le cache principal (5). Le cache principal peut avoir besoin d'éjecter un bloc nécessitant un *writeback* (6). S'il est nul, ce bloc peut être alloué dans le cache de zéros (7).
- Lors de la lecture d'un bloc nul, les deux caches peuvent contenir le bloc demandé. En cas de succès dans au moins un des caches, le bloc est retourné vers le niveau supérieur, figure 3.2(b). En cas d'échecs dans les deux caches, figure 3.2(d), le bloc

est demandé au niveau inférieur (3) ; lors de son retour (4), il passe dans le détecteur de blocs nuls ; comme il est nul, il est donc stocké dans le cache de zéros (5). Le cache de zéros ne contenant aucun bloc marqué comme *dirty*, aucun *writeback* n'est nécessaire.

Ainsi les informations de défaut des deux caches sont nécessaires pour propager le défaut au niveau inférieur. Pour ne pas risquer de dégrader les performances, le cache de zéros doit répondre plus vite que le cache principal. Cela impose un cache de zéros rapide avec une taille et une associativité inférieures ou égales à celles du cache principal.

3.1.3 Accès en écriture



(a) bloc non-nul (b) bloc nul

Figure 3.3 – Écriture d'un bloc nul/non-nul

Lors d'une l'écriture dans le *Zero-Content Augmented Cache*, la nouvelle donnée doit pouvoir être propagée aux niveaux inférieurs. Dans le but de garder le cache de zéros aussi petit et simple que possible, le bloc est stocké dans le cache principal, et si besoin invalidé dans le cache de zéros. Ainsi, le cache de zéros ne contient aucune donnée marquée comme *dirty*. Même lors de l'écriture d'un bloc nul, le bloc est écrit dans le cache principal pour être propagé.

Cependant, le détecteur de blocs nuls et le cache de zéros permettent de détecter une partie des blocs précédemment nuls réécrits nuls. Si le bloc est déjà présent dans le cache de zéros, l'écriture est ignorée car le cache de zéros ne contient aucune donnée

modifiée. Ce mécanisme permet de diminuer le nombre de *writebacks*, à la façon des *Silent Stores* [63] de *Lepak et Lipasti*.

La cohérence entre les différents processeurs est assurée par le cache principal. Ainsi, si un processeur demande l'usage exclusif d'un bloc, le cache de zéros doit invalider le bloc. De même, lorsqu'un autre processeur écrit un bloc non-nul, le cache de zéros doit l'invalider.

De façon optionnelle, les blocs nuls éjectés du cache principal peuvent passer par le détecteur de blocs nuls afin de les charger dans le cache de zéros.

Les accès en écriture sont représentés sur la figure 3.3 page précédente On considère un cache *write-allocate* :

- Lors de l'écriture d'un bloc non-nul, représentée figure 3.3(a), le bloc est écrit dans le cache principal (1) et il est invalidé (2), s'il est présent, dans le cache de zéros. Comme dans le cas de la lecture, si le cache principal doit effectuer un *writeback* (3), le bloc éjecté peut être alloué dans le cache de zéros s'il est nul (4).
- Lors de l'écriture d'un bloc nul, représentée figure 3.3(b), le bloc est recherché dans le cache de zéros. S'il est présent, l'écriture est ignorée. Sinon, il est écrit dans le cache principal (1). Dans ce cas aussi, si le cache principal doit effectuer un *writeback* (4), le bloc éjecté peut être alloué dans le cache de zéros s'il est nul (5).

3.1.4 Coût matériel

Le cache de zéros doit avoir un coût matériel faible. C'est un cache sectorisé. Cependant, la taille des secteurs est limitée à la taille des pages physiques du système. Utiliser des pages plus grandes ne serait pas pertinent car la localité spatiale n'est pas préservée au-delà d'une page physique lors de la traduction d'adresse virtuelle en adresse physique. Nous n'allons donc considérer que des secteurs d'au plus 8 Ko.

Le coût de stockage d'une entrée dans le cache de zéros est réparti [1] entre le *tag* et les $N = 2^n$ bits de validité (un par bloc dans le secteur). Soient A et $S = 2^s$ respectivement l'associativité et le nombre de *sets* du cache de zéros. Soit $B = 2^b$ la taille d'un bloc du cache et P la taille d'une adresse physique.

Dans ces conditions, chaque entrée coûte $(P - s - n - b) + N$ bits, soit un volume total de $A * S * (N + (P - s - n - b))$ bits pour un cache de zéros pouvant représenter jusqu'à $A * S * N * B$ octets.

Le tableau 3.2 page ci-contre illustre le volume occupé par un cache de zéros de 8 voies, associatif par ensemble pour des secteurs allant de 2 Ko à 8 Ko et des caches de zéros de 1 Mo à 128 Mo (c'est-à-dire des caches de zéros pouvant représenter jusqu'à 1 Mo à 128 Mo). Nous considérons des adresses physiques de 50 bits et des blocs de 64 octets.

1. Pour des raisons de simplicité, les bits de la politique de remplacement ne sont pas comptés.

Taille de Secteur	Volume de blocs nuls stockables dans le cache de zéros							
	1 Mo	2 Mo	4 Mo	8 Mo	16 Mo	32 Mo	64 Mo	128 Mo
2 Ko	4	7,8	15,5	30	60	120	240	493
4 Ko	3	5,9	11,7	23	46	91	180	356
8 Ko	2,5	4,9	9,9	20	39	77	154	306

Table 3.2 – Coût de stockage (en Ko) pour différentes tailles de secteurs et de caches de zéros

Ce tableau montre clairement qu'avec des secteurs de 4 ou 8 Ko, de grandes zones de blocs nuls peuvent être stockées pour un coût très raisonnable. Par exemple, avec des secteurs de 8 Ko, on peut représenter jusqu'à 4 Mo de blocs nuls avec seulement 10 Ko de stockage et jusqu'à 32 Mo de blocs nuls avec 77 Ko de stockage.

3.1.5 Consommation électrique

Comme le laisse supposer le taux de blocs nuls mesuré dans le chapitre 2, l'analyse de performance section 3.2 va montrer de grandes disparités dans l'efficacité du cache de zéros selon les applications. Pour les applications présentant un très faible taux de blocs nuls, l'accès en parallèle au cache de zéros est une dépense d'énergie. Elle ne se traduit par aucun gain de performance.

Ce gaspillage peut être évité en désactivant le cache de zéros. En utilisant un des détecteurs de blocs nuls et un compteur saturé, il est possible de surveiller le taux de blocs nuls. Lorsque ce taux est en dessous d'un certain seuil, le cache de zéros peut être désactivé. Le cache ne contenant aucun bloc modifié, cette désactivation est immédiate, aucun *writeback* n'est nécessaire. Lors de la réactivation du cache, il suffit de veiller à ce que toutes les lignes soient marquées comme invalides. Seuls le détecteur de blocs nuls et un compteur saturé restent actifs, mais leur consommation est infime comparée au cache principal.

3.1.6 Position du cache de zéros dans la hiérarchie mémoire

De nombreuses configurations sont envisageables. Le *Zero-Content Augmented Cache* peut être utilisé à tous les niveaux de la hiérarchie mémoire. Il est même possible de l'utiliser à plusieurs niveaux simultanément.

3.1.6.1 Au niveau du cache L1

L'utilisation d'un *Zero-Content Augmented Cache* de niveau 1 impose des contraintes en termes de latence et de taille.

La latence globale du *Zero-Content Augmented Cache* doit rester très proche de celle du cache principal. Elle sera légèrement supérieure à cause du temps nécessaire pour multiplexer les résultats. Le temps d'accès au cache de zéros doit donc être masqué par le cache principal. Au premier niveau de cache, la latence d'accès au cache principal est dominée par les *tags*. Il faut donc limiter le nombre d'entrées et l'associativité du cache de zéros pour rester dans le même ordre de complexité.

Dans la configuration choisie pour évaluer les performances, nous avons opté pour un cache DL1 de 32 Ko, associatif 4 voies avec des lignes de 64 octets. Cela représente 512 entrées. Le cache de zéros doit donc contenir moins de 512 entrées. Nous utiliserons donc un cache de zéros de 128 entrées, associatif 4 voies avec des secteurs de 8 Ko. D'après le tableau 3.2 page précédente, il occupe 2,5 Ko pour 1 Mo de blocs nuls.

Afin d'évaluer la latence induite par la comparaison des résultats du cache principal et du cache de zéros, nous avons utilisé Cacti [88]. Pour modéliser le *Zero-Content Augmented Cache*, nous avons choisi un cache de 64 Ko, 2 bancs, 8 voies. En effet, dans le pire cas, la complexité du cache peut doubler (le cache principal et le cache de zéros) mais sur 2 bancs. Cacti donne un temps d'accès de 483 ps, contre 451 ps pour le cache principal. Dans cette configuration, la latence supplémentaire est donc négligeable.

3.1.6.2 Au niveau du cache L2

L'utilisation d'un *Zero-Content Augmented Cache* de niveau 2 permet plus de liberté sur la taille. C'est un compromis taille-latence.

Dans la configuration choisie pour évaluer les performances, nous avons choisi un cache principal de 256 Ko, associatif 4 voies et un cache de zéros de 1024 secteurs de 8 Ko. D'après le tableau 3.2 page précédente, ce cache de zéros occupe 20 Ko pour 8 Mo de blocs nuls.

3.1.6.3 Au niveau du cache L3

L'utilisation d'un *Zero-Content Augmented Cache* de niveau 3 autorise un cache de zéros de grande taille. De plus la latence entre ce niveau et le niveau inférieur (la mémoire principale) est importante. Une réduction du nombre d'échecs peut plus facilement augmenter les performances.

Dans la configuration choisie pour évaluer les performances, nous avons pris un cache principal de 1 Mo, 8 voies, et un cache de zéros de 4096 entrées de 8 Ko. D'après le tableau 3.2 page précédente, ce dernier occupe 77 Ko pour 32 Mo de blocs nuls.

Dans le *Zero-Content Augmented Cache*, le cache principal et le cache de zéros sont accédés en parallèle. Or, souvent, afin d'économiser de l'énergie, les *tags* et les données du cache principal sont accédés séquentiellement, i.e. les données ne sont accédées que s'il y a un succès dans les *tags*. Le *Zero-Content Augmented Cache* peut donc être plus rapide que le cache principal pour un bloc nul.

3.1.6.4 À plusieurs niveaux simultanément

Il est aussi possible d'utiliser le *Zero-Content Augmented Cache* à plusieurs niveaux de la hiérarchie mémoire simultanément.

Une telle hiérarchie permet alors une optimisation qui réduit significativement le taux d'échecs dans le cache le plus rapide. Considérons, par exemple, deux *Zero-Content Augmented Cache*, le premier au niveau 1 et le second au niveau 2. Lors d'un succès dans le cache de zéros du niveau 2, et donc d'un échec au niveau 1, un nouveau secteur est alloué dans le cache de zéros du niveau 1. Ce secteur ne contient que le bit de validité du bloc. Il y aura un échec pour chaque accès suivant à un bloc de ce secteur. Une optimisation consiste à transférer le secteur complet au lieu du bloc nul. Un secteur de 8 Ko occupe 16 octets de stockage, soit moins qu'un bloc de 64 octets. Cela revient à faire un pré-chargement de tous les blocs d'un secteur du cache de zéros de niveau 1, diminuant ainsi le nombre d'échecs.

Lors de l'utilisation d'un tel mécanisme, des blocs pouvant être modifiés dans le cache principal mais pas encore propagés sont lus depuis le niveau inférieur. Le cache de zéros doit alors être considéré comme d'un niveau inférieur au cache principal. Ainsi, les deux caches peuvent être interrogés simultanément. Mais s'il y a un succès dans les deux caches, la réponse du cache de zéros est ignorée. Lors d'un *writeback* d'un bloc du cache principal, si ce bloc est non-nul le cache de zéros doit impérativement être mis à jour.

L'utilisation du *Zero-Content Augmented Cache* à plusieurs niveaux de la hiérarchie mémoire simultanément permet de réduire la latence d'accès à un bloc nul. Nous verrons cependant dans la section 3.2 que les gains de performances d'un *Zero-Content Augmented Cache* un peu plus rapide sont négligeables.

3.1.7 Politique de remplacement

Afin de gérer les secteurs dans le cache de zéros, une légère modification de la politique de remplacement est nécessaire au niveau du cache de zéros.

Un secteur est alloué à chaque fois qu'un bloc nul est accédé. Or, ce bloc peut être écrasé et ne rester nul qu'un instant. Dans ce cas, son bit de validité est mis à zéro. Cela peut conduire à la présence de secteurs sans aucun bit de validité positionné. Dans ce cas, le secteur est marqué comme invalide, et son emplacement est rendu disponible pour allouer un autre secteur.

3.1.8 Utilisation avec une mémoire compressée

Dans le chapitre 4, nous proposerons une architecture de mémoire compressée capable elle aussi de gérer efficacement les blocs nuls. Cette architecture pourra communiquer la liste des blocs nuls d'une page, à la façon d'un *Zero-Content Augmented Cache*, sur plusieurs niveaux, tel que décrit dans la section 3.1.6.4.

Lors d'un échec sur un bloc nul, le contrôleur de mémoire compressée (cf. section 1.4.2.4 page 34) peut communiquer un secteur qui sera alloué dans le cache de zéros.

Dans nos évaluations, nous considérerons que le contrôleur de mémoire compressée effectue un échec dans son cache interne et va chercher cette donnée en mémoire. Cette évaluation est, certes, pessimiste, mais il est fort probable que si le cache de zéros ne contient pas le secteur, alors le cache du contrôleur mémoire ne le contient pas pas non plus.

Ce pré-chargement depuis la mémoire peut être envisagé sans mémoire compressée, en utilisant une table résidant en mémoire. Celle-ci peut avoir une structure semblable à celle utilisée pour la pagination. La table occupe 1 bit/bloc, soit $1/512^{e}$ de l'espace. Le dernier niveau de cache, ou le contrôleur mémoire, doit être modifié pour que lors de chaque *writeback* le bit correspondant au bloc soit mis à jour. Cette table peut alors servir à charger des secteurs complets dans le cache de zéros.

3.2 Évaluation des performances

Dans cette section, nous allons évaluer les performances en termes de réduction du nombre d'échecs, de diminution de l'utilisation de la bande passante mémoire et d'augmentation du nombre d'instructions par cycle.

3.2.1 Infrastructure de simulation

Afin d'évaluer les performances du *Zero-Content Augmented Cache*, nous avons utilisé *Super Escalar Simulator* (Sesc) [73]. Il s'agit d'un simulateur très détaillé.

Il offre de nombreux avantages par rapport à SimpleScalarOOO [10], principalement au niveau de la hiérarchie mémoire. Dans Sesc, celle-ci est finement détaillée, les *Miss State Handling Register* [57, 78] sont présents, les caches sont pipelinés et les bus sont modélisés. Le cœur d'exécution est lui aussi simulé en détail avec une gestion de l'exécution dans le désordre.

Deux émulateurs utilisant le jeu d'instructions MIPS sont intégrés : l'un est nommé *Mint* et l'autre *Emul*. *Mint*, l'émulateur par défaut, effectue une émulation au niveau des appels à la bibliothèque C. Cependant, la compilation d'applications est très difficile, le format ELF étant modifié. Seul le *cross-compiler* fourni permet de compiler quelques applications. La bibliothèque C émulée contient de nombreux bugs empêchant l'exécution d'une part importante des SPEC CPU 2000 et 2006. L'émulateur *Emul* a été ajouté récemment à Sesc par *Milos Prvulovic*. Il émule au niveau des appels systèmes et résout la totalité des problèmes de *Mint*. Les résultats présentés ont tous été obtenus avec *Emul*.

Sesc s'exécute beaucoup plus vite que d'autres simulateurs comme SimpleScalarOOO [10], pour un niveau de détail supérieur. Il simule environ 500 KI/s, soit 24 heures pour cinquante milliards d'instructions.

Nous n'avons pas retenu l'utilisation de SimPoint [69] afin d'accélérer les simulations. SimPoint permet, d'après la structure d'un programme, d'extraire des morceaux représentatifs. Ces morceaux sont ensuite simulés et affectés d'un poids pour établir un résultat moyen sur toute l'application. Ce mécanisme est adapté aux éléments architecturaux ne nécessitant qu'un court temps de préchauffage (*warm-up*). Or, notre proposition permet de stocker jusqu'à 32 Mo de blocs nuls. Le temps de préchauffage est alors beaucoup trop important.

3.2.1.1 Applications simulées

Nous avons utilisé les applications des SPEC CPU 2000 et 2006, avec leurs jeux d'entrées *ref*. Une attention particulière a été donnée aux applications *255.vortex*, *481.wrf* et *482.sphinx3* dont le jeu d'entrées dépend de l'*endianness*. Malgré la correction de quelques erreurs dans l'émulateur, il n'a pas été possible de faire fonctionner *481.wrf* correctement. Celle-ci se termine prématurément. Nos expérimentations faites avec *Simics* dans le chapitre 4 montrent que *481.wrf* a un comportement très proche de *459.GemsFDTD*.

Afin de pouvoir tester toutes les applications des SPEC CPU 2000 et 2006, nous avons choisi de générer notre propre *cross-compiler* gérant C, C++ et Fortran. Il est basé sur *gcc 4.4*, *eglibc 2.12* et *binutils 2.20*. Il produit des binaires MIPS-IV en 32 bits. Les applications sont compilées avec le niveau d'optimisation `-O3`. Afin de garantir une reproductibilité des résultats, elles sont compilées en statique.

Les applications choisies sont celles que l'on a analysées dans le chapitre 2. Pour chacune de ces applications, la mesure du nombre d'accès et de la proportion de blocs nuls dans ces accès est représentée dans le tableau 2.1 de la page 38.

3.2.1.2 Architecture simulée

La hiérarchie mémoire simulée est celle décrite dans la section 3.1.6. Les autres paramètres importants de l'architecture choisie sont résumés dans le tableau 3.3 page suivante.

3.2.2 Position du cache de zéros dans la hiérarchie mémoire

Dans la section 3.1.6, nous avons discuté des différentes positions possibles du *Zero-Content Augmented Cache* au sein de la hiérarchie mémoire. Il s'agit principalement d'un compromis *grande taille* vs *faible latence*.

Les histogrammes de la figure 3.4 page 63 présentent l'IPC normalisé et la réduction du nombre d'accès mémoire pour les différentes positions possibles du *Zero-Content Augmented Cache*.

CPU		MIPS-IV, ABI O32, (Sesc avec libemul)
Decode, Issues, width		4
Retire width		5
Taille ROB		26 Issues + 48 entrées
Taille LSQ		10 Issues + 40 entrées
Prédicteur de branchements		O-GEHL [82], 64 Kbits, 6 cycles de pénalité par mispred.
L1 instructions		64 Ko, direct-map, 64 octets/bloc, 1 cycle
L1 données	Cache principal	32 Ko, 4 voies, ligne de 64 octets, LRU, 1 cycle, WB
	Cache de zéros (Optionnel)	128 entrées, secteurs de 8 Ko 4 voies, LRU, 1 Mo (pour un coût de 2, 5 Ko)
L2 unifié	Cache principal	256 Ko, 4 voies, ligne de 64 octets, LRU, 11 cycles, WB
	Cache de zéros (Optionnel)	1024 entrées, secteurs de 8 Ko 4 voies, LRU, 8 Mo (pour un coût de 20 Ko)
L3 unifié	Cache principal	1 Mo, 8 voies, ligne de 64 octets, LRU, 30 cycles, 16 B/cycle, WB
	Cache de zéros (Optionnel)	4096 entrées, secteurs de 8 Ko 4 voies, LRU, 32 Mo (pour un coût de 78 Ko)
Mémoire principale		500 cycles, 16 octets/cycle

Table 3.3 – Configuration du simulateur Sesc.

3.2.2.1 Zero-Content Augmented Cache placé en L1, L2 ou L3

Les trois premières barres des histogrammes de la figure 3.4 page ci-contre représentent un *Zero-Content Augmented Cache* positionné en L1, L2 ou L3. Sans surprise, plus le *Zero-Content Augmented Cache* est large, meilleure est la réduction du nombre d'échecs.

Sur la figure représentant la réduction du nombre d'échecs L3, on remarque que pour quelques applications un *Zero-Content Augmented Cache* placé en L1 est suffisant. C'est le cas de *435.gromacs*, *465.tonto* et *482.sphinx3*. Pour *401.bzip2*, *447.dealII* et *459.GemsFDTD*, un cache de zéros L2 est nécessaire. *416.gamess*, *434.zeusmp* et *437.leslie3d* nécessitent un *Zero-Content Augmented Cache* placé en L3.

Sur la figure représentant les performances en IPC normalisé [2], les applications qui se contentaient d'un *Zero-Content Augmented Cache* placé en L1 n'affiche de gain de performance supérieur en L1 qu'en L2 ou L3. Le compromis *grande taille* vs *faible latence* penche en faveur d'un cache de zéros de grande taille en L3.

2. L'IPC normalisé est défini comme l'IPC avec ZCA Cache divisé par l'IPC sans ZCA Cache.

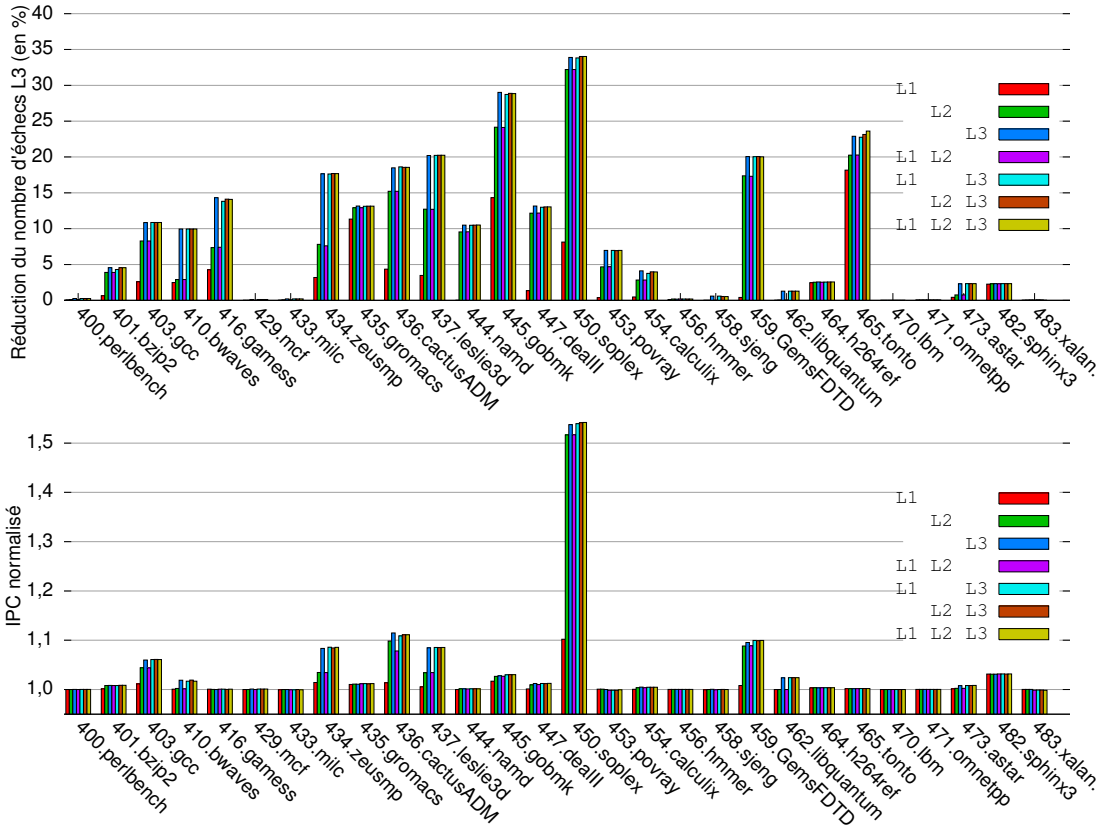


Figure 3.4 – Réduction du nombre d'échecs L3 (en %) et IPC normalisé pour différentes positions du *Zero-Content Augmented Cache* dans la hiérarchie mémoire.

3.2.2.2 Zero-Content Augmented Cache placé à plusieurs niveaux

Les quatre dernières barres de la figure 3.4 représentent l'utilisation de plusieurs *Zero-Content Augmented Cache* au sein de la hiérarchie mémoire. Le mécanisme de passage de secteurs décrit dans la section 3.1.6.4 est présent. Ainsi, "L1 L3" désigne une hiérarchie mémoire comportant deux *Zero-Content Augmented Cache* placés en L1 et en L3, le cache L2 étant un cache conventionnel.

Lorsque des *Zero-Content Augmented Caches* sont utilisés sur plusieurs niveaux, la réduction du nombre d'échecs est sensiblement identique à celle du plus grand des *Zero-Content Augmented Cache*. Ainsi, une configuration "L1 L2" aura presque le même nombre d'échecs qu'une configuration "L2". De même, les configurations "L1 L3", "L2 L3" et "L1 L2 L3" se comportent comme une "L3".

Presque aucun gain en termes d'IPC n'est visible en utilisant des *Zero-Content Augmented Caches* à plusieurs niveaux. Le compromis *grande taille* vs *faible latence* penche ici aussi en faveur d'un cache de zéros de grande taille.

3.2.3 Zero-Content Augmented Cache au niveau L3

Comme nous venons de le voir, le *Zero-Content Augmented Cache* placé au L3 est suffisant pour capturer la majeure partie des gains potentiels. Nous allons analyser en détail cette configuration.

	Configuration de Base				avec un ZCA L3			
					IPC	Réd.	Bus Mémoire	
	Nul	MPKI	IPC	Latence	Norm.	Lat.	Lect.	Ecr.
Application	(%)	(cycles)	(i/c)	(cycles)		(%)	(%)	(%)
400.perlbench	6	0,4	1,12	3	1,00	0	0	0
401.bzip2	8	2,0	0,91	10	1,01	4	5	2
403.gcc	20	5,4	0,51	24	1,06	18	11	49
410.bwaves	30	3,7	1,31	14	1,02	13	10	39
416.gamess	46	0,0	1,70	2	1,00	0	14	40
429.mcf	1	30,4	0,15	58	1,00	0	0	0
433.milc	7	18,1	0,38	87	1,00	0	0	1
434.zeusmp	73	4,5	0,87	21	1,08	15	18	72
435.gromacs	9	0,5	1,29	4	1,01	7	13	3
436.cactusADM	59	3,4	1,22	6	1,11	8	18	21
437.leslie3d	20	10,8	0,58	67	1,08	18	20	18
444.namd	23	0,1	1,34	3	1,00	2	10	5
445.gobmk	24	0,4	0,93	4	1,03	18	29	26
447.dealII	18	1,0	1,17	10	1,01	8	13	26
450.soplex	34	16,8	0,28	94	1,54	47	34	49
453.povray	42	0,0	1,65	2	1,00	0	7	18
454.calculix	12	0,5	1,02	5	1,00	1	4	6
456.hmmer	0	1,9	1,23	8	1,00	0	0	0
458.sjeng	47	0,4	0,90	4	1,00	0	1	42
459.GemsFDTD	97	13,6	0,55	33	1,10	17	20	99
462.libquantum	2	18,8	0,39	208	1,02	1	1	4
464.h264ref	2	0,8	1,28	7	1,00	1	3	3
465.tonto	16	0,0	1,82	2	1,00	1	23	21
470.lbm	0	23,2	0,64	87	1,00	0	0	0
471.omnetpp	0	20,5	0,46	52	1,00	0	0	0
473.astar	3	3,9	0,42	16	1,01	3	2	2
482.sphinx3	4	9,5	0,46	72	1,03	3	2	33
483.xalancbmk	0	12,4	0,31	46	1,00	0	0	0

Table 3.4 – Performances du ZCA Cache en L3. À gauche la configuration initiale, à droite la configuration avec un ZCA cache L3 avec l'IPC normalisé, la réduction de la latence d'accès à la mémoire, et les réductions des lectures et écritures sur le bus mémoire.

Le tableau 3.4 page ci-contre présente dans sa partie gauche les caractéristiques des applications de la suite SPEC CPU 2006 pour la configuration initiale. Sont ainsi représentés : le taux de blocs nuls dans les échecs du dernier niveau de cache, le nombre d'échecs pour mille instructions, l'IPC et la latence moyenne de la hiérarchie mémoire complète. La partie droite représente quant à elle l'effet du *Zero-Content Augmented Cache*. Sont représentés : l'IPC normalisé, la réduction de la latence de la hiérarchie mémoire, et la réduction de l'occupation du bus mémoire. Ainsi, la réduction du nombre d'échecs correspond donc à la réduction du nombre de lectures sur le bus mémoire.

Dix applications de la suite SPEC CPU 2006 affichent une amélioration de l'IPC supérieure ou égale à 2%. Ce nombre est de treize sur vingt-quatre pour la suite SPEC CPU 2000. Les applications accédant à de nombreux blocs nuls sont celles qui bénéficient le plus du *Zero-Content Augmented Cache*. Pour ces applications, le *Zero-Content Augmented Cache* capture la localité temporelle des blocs nuls : ils sont lus nuls, et réutilisés. L'utilisation du bus mémoire est fortement réduite pour la majorité des applications faisant de nombreux échecs sur des blocs nuls.

L'application *450.soplex* montre une augmentation de l'IPC de 54%. Ce gain s'explique par un nombre d'échecs important dans la configuration de base. La latence moyenne de la hiérarchie mémoire de 94 cycles par accès est l'une des plus élevées. Le *Zero-Content Augmented Cache* placé en L3 permet presque de diviser par deux cette latence (-47%) grâce à une réduction de 34% du nombre d'échecs du dernier niveau de cache.

Notre proposition permet d'améliorer significativement les performances lorsque la hiérarchie mémoire subie une forte charge. C'est le cas pour l'application *436.cactusADM* que nous avons analysée à la section 2.3.5 page 47. Cette application présente une très faible latence moyenne de 6 cycles. Cependant, 55% des instructions sont des accès mémoire. La réduction de 18% du nombre d'échecs et ainsi de 8% de la latence se traduit directement dans l'IPC par un gain de 11%.

D'autres applications affichent un taux de blocs nuls assez élevé, mais une très faible augmentation des performances et une faible réduction du nombre d'échecs. Cela est dû à des applications lisant ou écrivant des blocs nuls, mais les réécrivant non-nuls rapidement. C'est le cas de *416.gamess* et *444.namd*. Pour ces applications, l'allocation dans le cache principal est simplement retardée.

Pour d'autres applications, le cache de zéros ne permet pas de stocker assez de blocs nuls pour obtenir une réduction du nombre d'échecs à la hauteur du taux de blocs nuls. C'est le cas pour *434.zeusmp* et *458.sjeng*. *434.zeusmp* présente un taux de blocs nuls de 73% dans les échecs du dernier niveau de cache, mais une réduction de « seulement » 18% du nombre d'échecs avec un *Zero-Content Augmented Cache*. Un cache de zéros deux fois plus gros permet d'atteindre une réduction du nombre d'échecs de 37%.

3.2.3.1 Composition des échecs

Le *Zero-Content Augmented Cache* stocke les blocs nuls à l'extérieur du cache principal. Cela va permettre d'une part d'obtenir un meilleur taux de succès sur les blocs nuls et d'autre part de libérer de la place dans le cache principal.

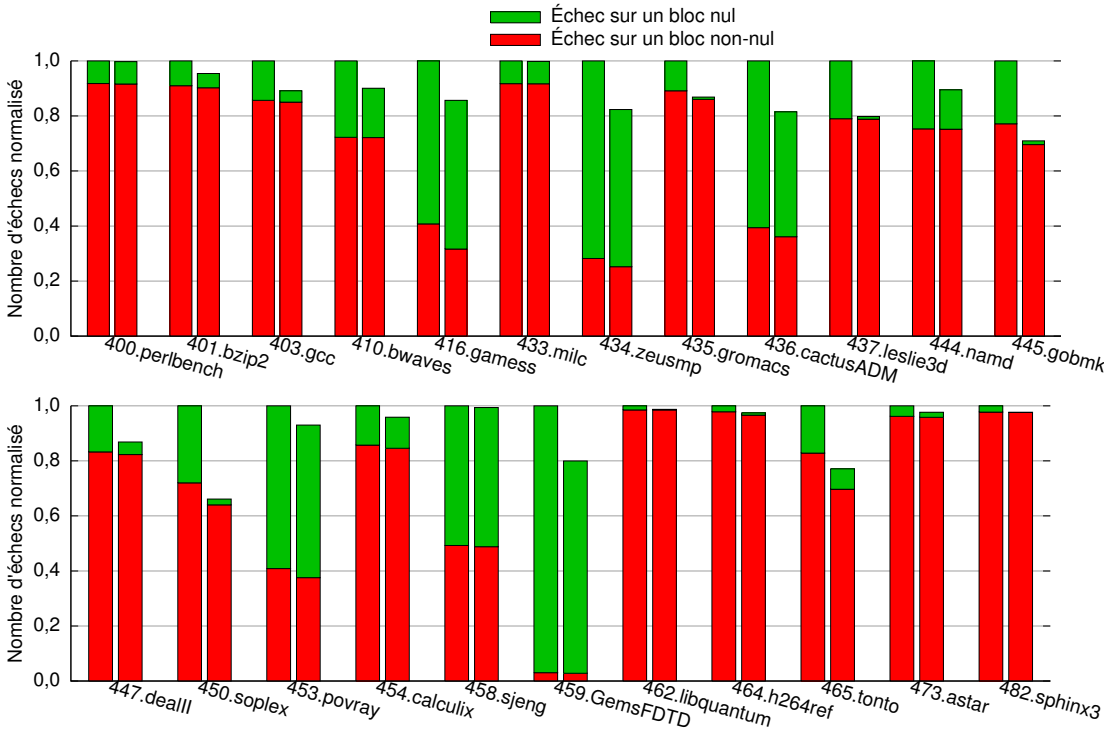


Figure 3.5 – Taux de blocs nuls dans les échecs pour une configuration sans *ZCA Cache* (barre de gauche) et avec un *ZCA Cache* (barre de droite).

La figure 3.5 illustre ces deux phénomènes. Pour chaque application deux colonnes sont représentées. Celle de gauche correspond à une architecture sans *Zero-Content Augmented Cache* et celle de droite à une architecture avec *Zero-Content Augmented Cache*. Les applications *429.mcf*, *456.hmmer*, *470.lbm*, *471.omnetpp* et *483.xalancbmk* accèdent à moins de 1% de blocs nuls. Elles ne sont pas représentées.

Pour les applications *403.gcc*, *435.gromacs*, *437.leslie3d*, *445.gobmk*, *447.dealII* et *450.soplex*, notre proposition a permis d'éliminer la plupart des blocs nuls des échecs.

Pour *416.gamess*, *434.zeusmp*, *436.cactusADM*, *453.povray*, *458.sjeng* et *459.GemsFDTD*, la part de blocs nuls dans les échecs reste importante. Un cache de zéros plus grand permet d'éliminer la majorité de ces échecs. Nous verrons dans la section 3.2.3.3 qu'une l'utilisation conjointe du *Zero-Content Augmented Cache* et d'une mémoire compressée permet de réduire significativement le nombre d'échecs.

Les applications *416.gamess*, *445.gobmk*, *450.soplex* et *465.tonto* présentent une baisse du nombre d'échecs sur des blocs non-nuls. Le cache de zéros a donc permis de libérer de l'espace dans le cache principal.

3.2.3.2 Évolution des performances au cours de la simulation

Dans cette section, nous allons mesurer l'évolution au cours de la simulation de l'IPC normalisé des applications de la suite SPEC CPU 2006. Cela va nous permettre une nouvelle fois de nous assurer que les gains de performances ne sont pas simplement des phénomènes d'initialisation, mais qu'ils existent tout au long de la simulation.

La figure 3.6 page suivante présente l'IPC normalisé des applications de la suite SPEC CPU 2006. Il s'agit d'une valeur moyenne sur des tranches d'un milliard d'instructions. Cette figure est à mettre en parallèle avec l'évolution du taux de blocs nuls présentée dans la figure 2.1 page 41 du chapitre 2.

L'évolution de l'IPC normalisé est très fortement corrélée avec l'évolution du taux de blocs nuls. Les applications qui présentent un taux élevé et stable de blocs nuls ont un IPC normalisé stable. C'est le cas de *434.zeusmp*, *436.cactusADM*, *437.leslie3d*, *445.gobmk* et *459.GemsFDTD*. Inversement, l'application *403.gcc*, dont le taux de blocs nuls varie fortement, présente elle aussi de fortes variations de l'IPC.

Les phases d'initialisation sont visibles sur quelques applications, mais celles-ci n'excèdent jamais deux milliards d'instructions. Le comportement des applications durant les phases d'initialisation est variable. Certaines applications affichent un *speedup*. C'est le cas de *462.libquantum* avec un IPC normalisé pendant l'initialisation $3,3$, et de $1,0$ après. D'autres applications affichent au contraire un faible gain durant cette phase. C'est les cas de *450.soplex* : pendant l'initialisation, l'IPC normalisé est de $1,01$ alors qu'il est de $1,7$ juste après.

3.2.3.3 Évaluation avec une mémoire compressée

Comme nous l'avons précisé à la section 3.1.8, l'utilisation conjointe du *Zero-Content Augmented Cache* et d'une mémoire compressée (ou d'un mécanisme stockant un bit par bloc) permet de pré-charger tous les blocs nuls d'un secteur (page de 8 Ko) depuis la mémoire.

La figure 3.7 représente les mesures de l'IPC normalisé et de la réduction du nombre d'échecs pour un *Zero-Content Augmented Cache* avec ou sans mémoire compressée [3]. Pour toutes les applications, les résultats sont bien meilleurs avec un pré-chargement depuis la mémoire. Concernant la réduction du nombre d'échecs, les applications qui en bénéficient le plus sont *416.gamess*, *434.zeusmp*, *453.povray*, *458.sjeng* et *459.GemsFDTD*.

La figure 3.8 page 70 reprend la figure 3.5 page précédente en ajoutant une troisième colonne représentant l'utilisation conjointe d'un *Zero-Content Augmented Cache*

3. La structure de la mémoire compressée est décrite dans le chapitre suivant.

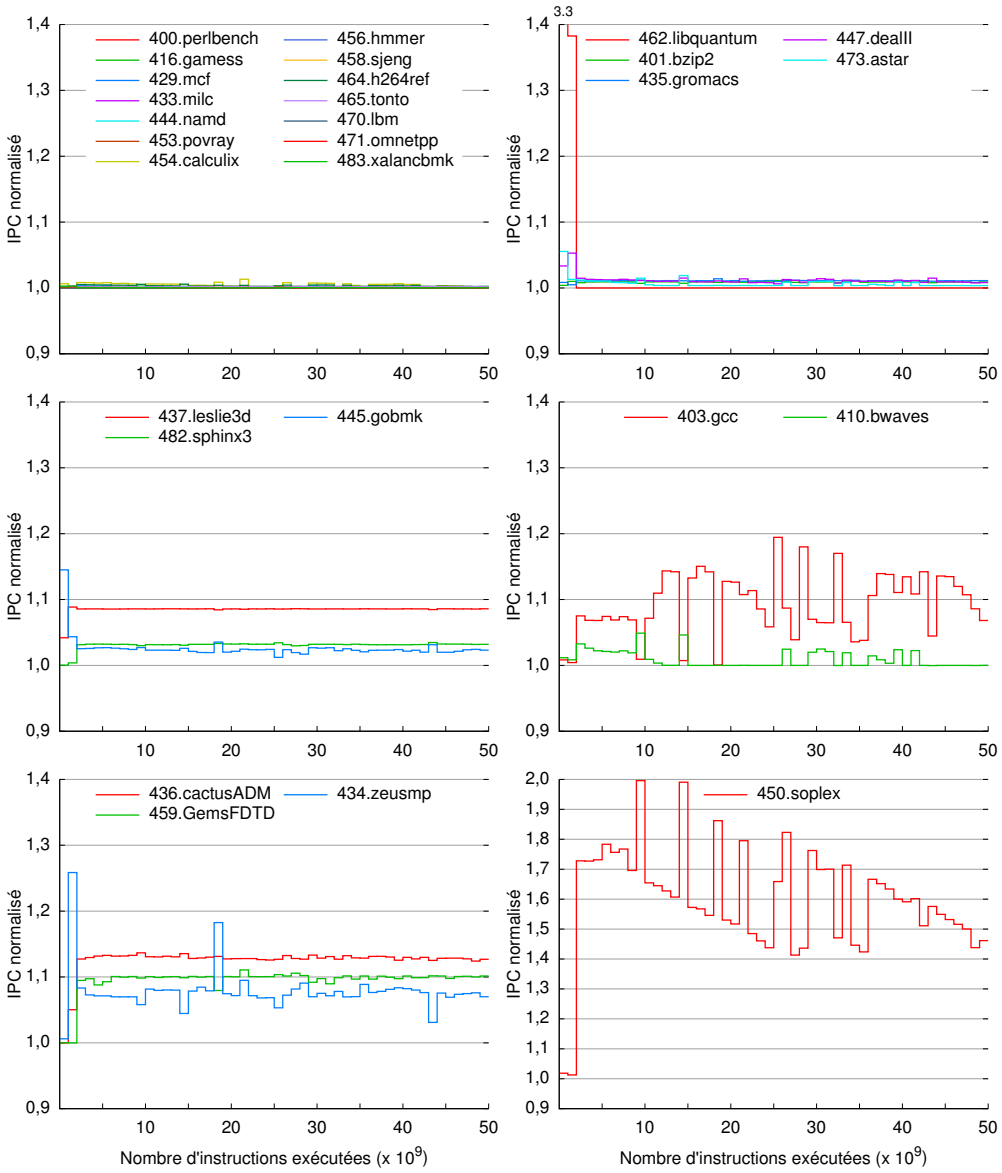


Figure 3.6 – Évolution de l'IPC normalisé sur 50.10^9 instructions.

et d'une mémoire compressée. Les applications qui bénéficient le plus de l'utilisation du mécanisme de pré-chargement depuis la mémoire sont celles qui conservaient un fort taux de blocs nuls malgré l'utilisation du *Zero-Content Augmented Cache* seul.

Par exemple, pour les applications *434.zeusmp* et *459.GemsFDTD*, l'utilisation du pré-chargement depuis la mémoire a permis d'éliminer la plupart des échecs sur des blocs nuls. En ce qui concerne l'IPC, le *speedup* est lui aussi important. L'application *434.zeusmp* passe d'un IPC normalisé de $1,08$ à $1,40$. Quant à *459.GemsFDTD*, le *speedup* est plus important encore : l' IPC normalisé passe de $1,10$ à $2,62$.

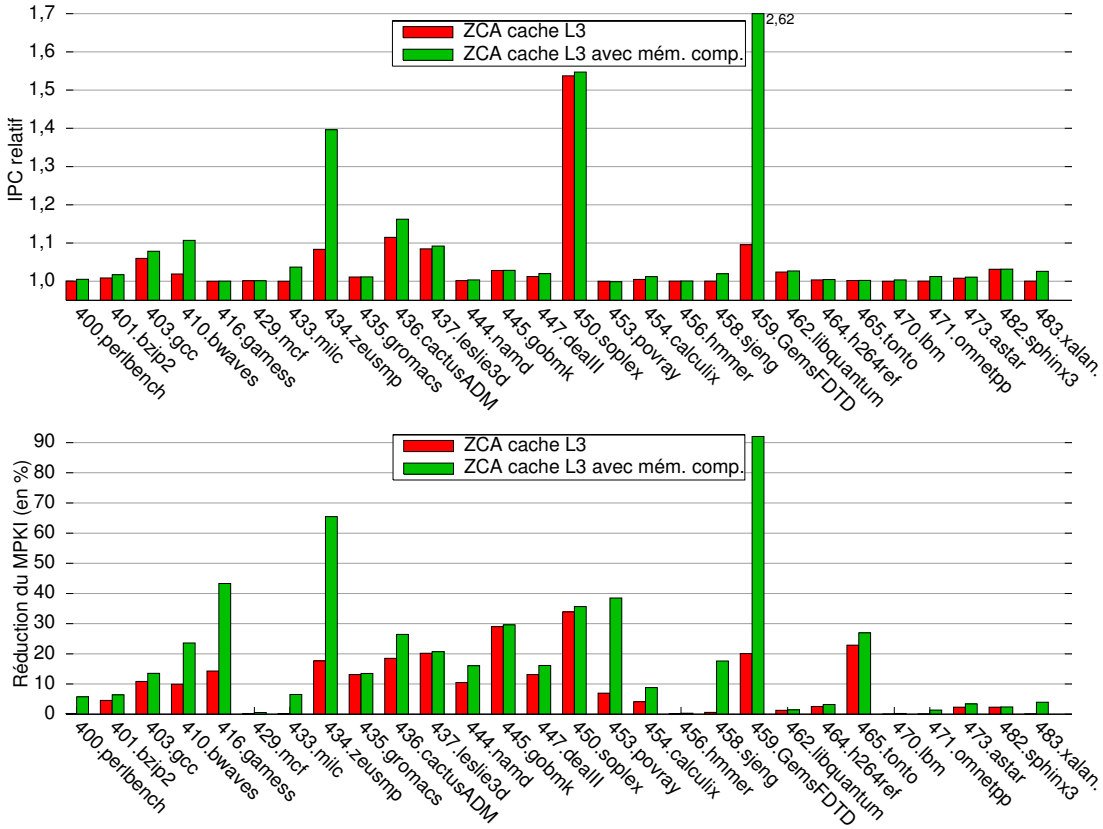


Figure 3.7 – IPC normalisé et réduction du MPKI (en %) pour un *Zero-Content Augmented Cache* en L3 avec ou sans mémoire compressée.

3.2.3.4 Localité spatiale des blocs nuls

Dans la section 3.1, nous avons proposé d'utiliser un cache sectorisé en supposant une forte localité spatiale des blocs nuls. En effet, les secteurs doivent contenir de nombreux blocs nuls pour être efficaces.

La figure 3.9 page suivante présente le taux de remplissage moyen des secteurs. Un taux de 100% signifie que tous les secteurs valides contiennent cent vingt-huit blocs nuls. Ce taux est un taux moyen, il est obtenu à partir de mesures effectuées tous les milliards d'instructions. Afin de réellement mesurer la localité spatiale, ce taux est obtenu avec le mécanisme de pré-chargement depuis la mémoire. Celle-ci permet de s'affranchir du temps de remplissage des secteurs.

Le taux de blocs nuls par secteur est supérieur à 50% pour la majorité des applications accédant à de nombreux blocs nuls. Les seules exceptions sont *410.bwaves* et *453.povray*. Pour ces deux applications, les blocs nuls sont répartis au sein de pages majoritairement non-nulles. Les applications *434.zeusmp*, *459.GemsFDTD* et *462.libquantum* présentent un taux de blocs nuls dans les secteurs compris entre 95 et 99%.

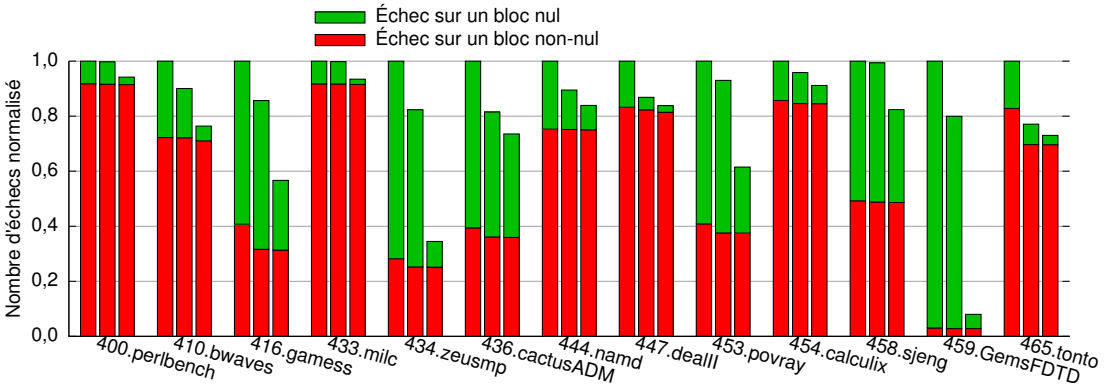


Figure 3.8 – Taux de blocs nuls dans les échecs sans *ZCA Cache*, avec *ZCA Cache* et avec *ZCA Cache* et mémoire compressée.

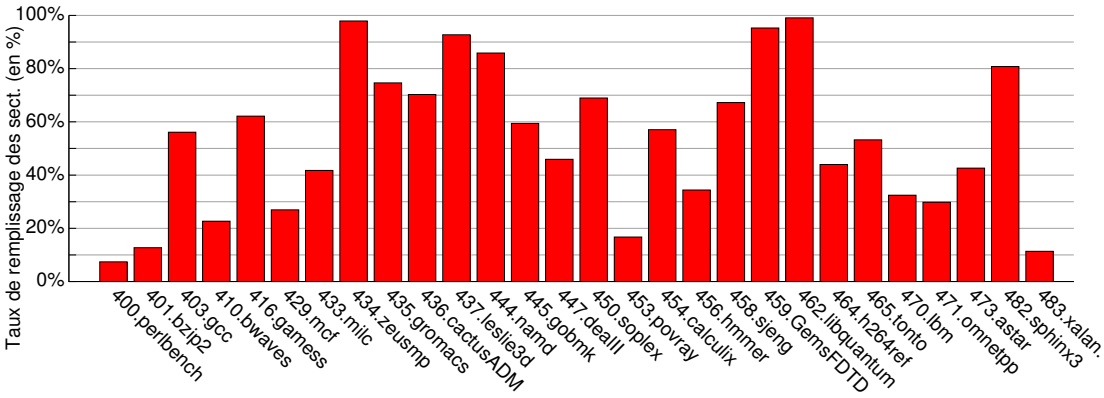


Figure 3.9 – Taux moyen de remplissage des secteurs (%)

L'utilisation de secteurs de 4 ou 8 Ko est donc pleinement justifiée, car la localité spatiale des blocs nuls est suffisamment importante.

3.3 Fusion du cache de zéros au sein du cache principal

Dans l'architecture décrite dans les sections précédentes, le cache de zéros occupe un espace réservé situé à l'extérieur du cache. Cet espace est alloué statiquement lors de la conception et ne peut être modifié pour s'adapter à l'application. Pour les applications n'exhibant pas un taux suffisant de blocs nuls, cet espace est alors perdu. Utiliser un cache commun pouvant stocker à la fois des secteurs du cache de zéros et des lignes permettrait d'exploiter au mieux la surface allouée.

3.3.1 Le cas des TLB

Cette problématique est très proche de ce qui se passe dans la MMU d'un processeur pouvant gérer plusieurs tailles de pages. La majorité des processeurs supporte plusieurs tailles. Par exemple, les processeurs ARM supportent 2 tailles : 4 Ko et 64 Ko, les x86 3 tailles : 4 Ko, 2 ou 4 Mo [4] et 1 Go [5], les processeurs Alpha 4 tailles : 8 Ko, 64 Ko, 512 Ko et 4 Mo et l'IA64 supporte 8 tailles : 256 Ko, 1 Mo, 4 Mo, 16 Mo, 64 Mo, 256 Mo, 1 Go, 4 Go.

Le *Translation Lookaside Buffer*, ou TLB, est un cache permettant d'accélérer la traduction d'adresses virtuelles en adresses physiques. Il associe à une page virtuelle une page physique. Utiliser plusieurs tailles de pages permet de diminuer le nombre d'entrées, et par conséquent le nombre de défauts. Les entrées de TLB pour les différents types de pages ont la même taille, mais leurs adresses n'ont pas le même format (figure 3.10). Il faut donc soit utiliser un TLB totalement associatif, soit utiliser un TLB par taille de pages [87, 86].

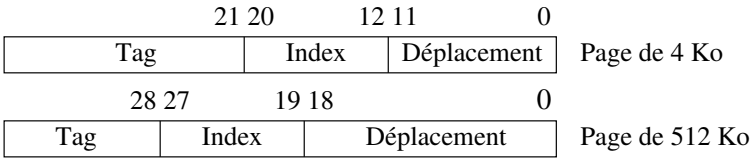


Figure 3.10 – Exemple du format d'adresse avec des pages de 4 et 512 Ko, et 512 entrées dans le TLB.

Utiliser un TLB totalement associatif limite considérablement sa taille. En effet, le TLB doit être très rapide (entre 0, 5 et 1 cycle pour le premier niveau), car sa latence est souvent masquée par l'utilisation d'un cache indexé virtuellement mais possédant des *tags* physiques. Le TLB est interrogé pendant que les différents *tags* et données du *set* correspondant à l'index virtuel sont ramenés dans des tampons.

Utiliser des TLB séparés n'est pas optimal. Pour tirer parti au mieux de cette fragmentation, les applications doivent utiliser tous les types de pages tout au long de l'exécution. Dans le cas d'un processeur gérant quatre tailles de page et d'une application n'utilisant que des petites pages, un seul des quatre TLB sera utilisé.

Utiliser un cache associatif par *set* unique pour les différentes tailles est une mauvaise idée [87]. A cause de leurs formats différents, plusieurs façons d'indexer les *sets*, sont possibles. Prenons l'exemple de la figure 3.10, c'est-à-dire d'un TLB de 512 sets gérant des pages de 4 et 512 Ko. Trois possibilités existent pour choisir l'index :

– *Utiliser les bits 20 à 12 comme index :* il n'y a plus aucun intérêt à utiliser de grandes pages. Une même grande page peut se retrouver sur plusieurs *sets* selon

4. 4 Mo avec Page Size Extension (PSE) et sans Physical Address Extension (PAE), et 2 Mo avec PSE et PAE

5. *long mode* sur les processeurs 64 bits

les bits 18 à 12. il y aura donc de multiples copies de la même entrée.
- *Utiliser les bits 27 à 19 comme index :* de très nombreux conflits apparaissent, 128 petites pages consécutives sont projetées sur le même *set*.
- *Utiliser les bits 27 à 19 pour les grandes pages et 20 à 12 pour les petites comme index :* plusieurs accès peuvent être nécessaires. De plus, lors de chaque succès, il faut vérifier le type de la page. Cette latence sur chaque accès dégrade plus les performances que l'utilisation de grandes pages n'apporte de gain éventuels [87].

Une proposition de Seznec [81] permet de résoudre ce problème. Dans cette proposition, il suggère de réserver des voies au sein d'un *set* pour certaines tailles de pages. Le choix des voies réservées dépend de bits pris dans le *tag*. Ainsi, cette répartition ne dépend pas du *set* mais du *tag*. Un même *set* n'aura pas toujours les mêmes voies assignées aux mêmes tailles de pages. Aucune ligne du cache ne stocke une taille précise de page. Grâce à une partie des bits d'adresse du bloc, on peut déterminer le type de page.

Ainsi cette fusion des différents caches de TLB au sein d'un même cache se fait au détriment de l'associativité. Il est cependant proposé d'utiliser un *Skewed Cache* [79] pour compenser cette perte.

3.3.2 Architecture du cache de zéros unifié

Nous adaptons ici la méthode proposée par Seznec [81], permettant de fusionner l'accès à des pages de tailles différentes dans le TLB, pour fusionner le cache principal et le cache de zéros.

3.3.2.1 Terminologie

Les deux caches étant fusionnés, il est important de faire la distinction entre les différents contenus possibles. Une *ligne* représente un emplacement dans le cache auquel est associé un *tag*. Cette ligne peut contenir soit un *bloc*, c'est-à-dire un morceau de donnée de 64 octets dans la plupart des caches, soit un *secteur*, c'est-à-dire une suite de bits correspondant à des booléens pour indiquer des blocs nuls ou non-nuls.

3.3.2.2 Structure

La structure du cache de zéros unifié est décrite dans la figure 3.11 page suivante. Elle est presque la même que celle décrite précédemment. La seule différence vient de la fusion du cache principal et du cache de zéros au sein d'un même cache. Les accès en lecture et en écriture se déroulent de façon exactement identique à ce qui a été décrit dans les sections 3.1.2 et 3.1.3 pages 53 et 55. Deux choix sont possibles : ajouter des ports au cache pour rendre les deux accès simultanés ou séquentialiser les accès. Dans ce dernier cas, la priorité est donnée au bloc de données, puis ensuite aux secteurs de zéros. En lecture, cela ne modifie pas la latence d'un succès sur un bloc, mais cela augmente légèrement la latence d'un succès dans un secteur de blocs nuls et d'un échec. Pour

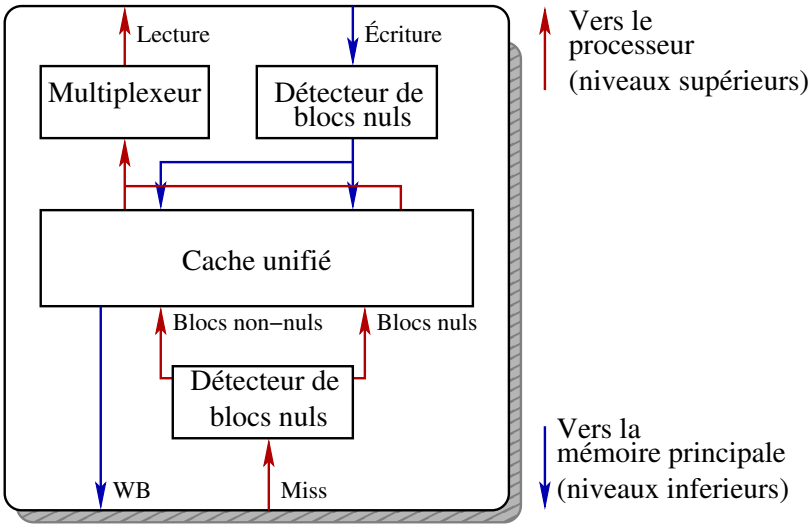


FIGURE 3.11 – Structure du Zero-Content Augmented Cache Unifié

l'écriture, la latence s'en retrouve elle aussi légèrement augmentée. Cependant, ces accès ne se situent pas sur le chemin d'accès critique.

La taille maximale d'un secteur est elle aussi limitée, mais ce n'est pas un problème. En effet, si l'on note B la taille d'une ligne de cache en octets, la taille maximale d'un secteur est de $64 \times 8B$, soit pour des lignes de 64 octets une taille maximale de secteur de 32 Ko.

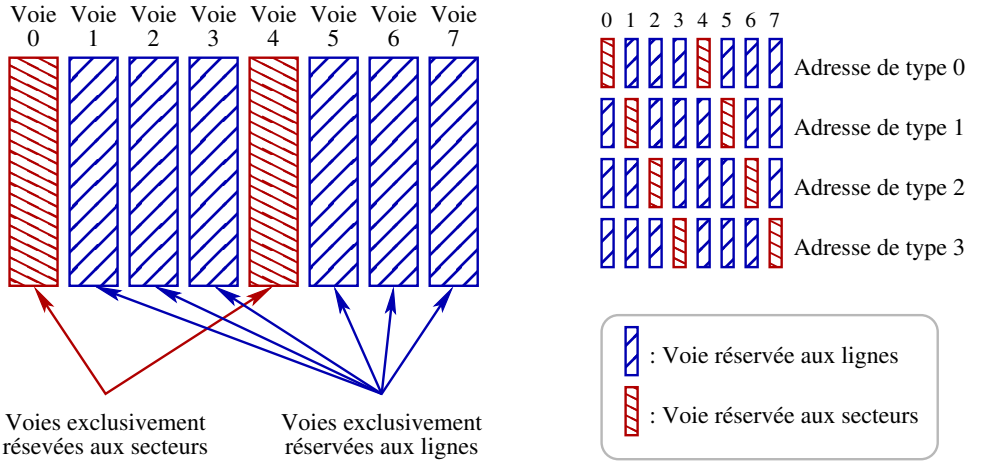
3.3.2.3 Répartition des voies

Pour chaque adresse, les voies du set sont réparties en lignes stockant des blocs ou des secteurs. On définit donc l'associativité du cache par 3 valeurs : l'associativité globale a, l'associativité du cache principal a_p et l'associativité du cache de zéros a_{zc}. Ainsi $a = a_p + a_{zc}$.

La figure 3.12(a) page suivante illustre cette répartition pour un cache défini par $a = 8$, $a_p = 6$ et $a_{zc} = 2$. Dans ce cache, pour chaque adresse, deux lignes sur les huit que comprend le set sont réservées pour stocker des secteurs. Les six lignes restantes sont réservées aux blocs. Ainsi, selon la voie sur laquelle le succès de cache a lieu, il est possible de déduire s'il s'agit d'un bloc ou d'un secteur.

On souhaite que toutes les lignes puissent contenir soit un bloc soit un secteur. Cela fait au minimum $min(a_p, a_{zc})/a = 4$ types d'adresses, ils sont représentés sur la figure 3.12(b) page suivante. Ainsi, une application n'utilisant que des secteurs ou que des blocs pourra tout de même utiliser toutes les lignes du cache au prix d'un sacrifice sur l'associativité.

Afin de sélectionner le type d'adresse, $log2(min(a_p, a_{zc})/a) = 2$ bits du *tag* sont nécessaires. Ils sont représentés par *s* sur la figure 3.13 page suivante. Afin de garantir



(a) Répartition des voies d'un set pour une adresse donnée de type 0.

(b) Différentes répartitions pour un set donné selon le type de l'adresse.

Figure 3.12 – Répartition des voies dans le cache de zéros unifié.

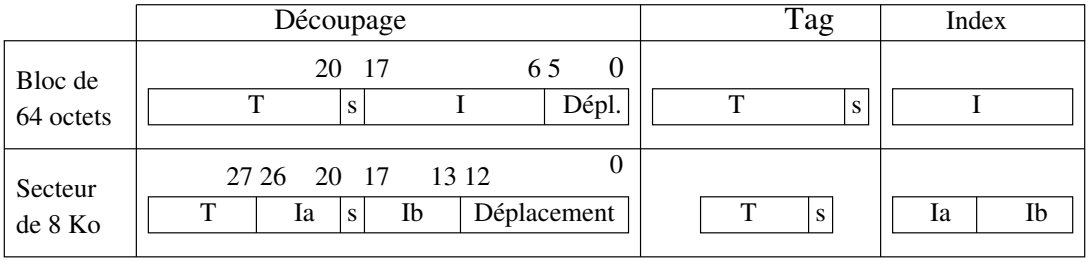


Figure 3.13 – Exemple du format d'adresse avec un cache de 1 Mo, 8 voies, des blocs de 64 octets et des secteurs de 8 Ko.

le maximum de mouvements, les bits de poids faibles du *tag* des blocs sont de bons candidats. Ces bits risquent de se retrouver dans l'index du secteur. Il faut dans ce cas découper en deux l'index et décaler la partie supérieure. Le trou ainsi créé est rajouté au *tag* (figure 3.13). Ce trou n'étant pas dans les poids faibles de l'index, il n'a que peu de chances d'augmenter le nombre de conflits, et ce grâce à la localité spatiale.

Aucune ligne n'étant réservée à des blocs ou à des secteurs, la quantité de blocs nuls stockables dans ce cache est très importante. Par exemple, le *Zero-Content Augmented Cache* L3 choisi dans la section 3.1 peut stocker 32 Mo de blocs nuls. Un *Zero-Content Augmented Cache* unifié de 1 Mo peut stocker au maximum $1024 * 1024/64 = 16384$ secteurs de 32 Ko, soit 512 Mo.

3.3.2.4 Politique de remplacement

Le cache de zéros unifié va contenir deux types de données : des blocs et des secteurs. Ces deux types de données vont entrer en concurrence au sein de *sets* du cache. Les lignes peuvent toutes contenir les deux types de données. De ce fait, un secteur peut être éjecté lors de l'allocation d'un bloc (dans les voies réservées aux blocs pour cette adresse). Inversement, lors de l'allocation d'un secteur, un bloc peut être éjecté. Or, les blocs et les secteurs ne vont pas avoir les mêmes fréquences d'utilisation, ni le même surcoût en cas de mauvais choix lors de l'éjection.

Lors de l'allocation d'un secteur, tous les blocs du secteur sont marqués comme non-nuls, à l'exception du bloc nul ayant provoqué l'allocation. Il faudra un échec sur chaque bloc nul pour le marquer comme tel dans le secteur. Ainsi, le coût de l'éjection d'un secteur est beaucoup plus important que celui d'un simple bloc.

La politique LRU nécessite d'être adaptée pour en tenir compte. De même, la politique de remplacement idéale définie par *Belady* [11] ne peut plus être appliquée.

Nous avons choisi d'adapter LRU en appliquant un coefficient p aux secteurs pour éviter leurs éjections. Lors de la sélection du contenu de la ligne en position LRU, s'il s'agit d'un secteur, ce bloc est n'éjecté qu'une fois sur p. Plusieurs choix sont possibles pour la valeur de p :

- On peut choisir p égal à une valeur prédéfinie, constante lors de l'exécution. Ce choix est le plus simple à mettre en œuvre. Cependant, la meilleure valeur de p va dépendre du comportement de l'application, et de la phase de l'exécution.
- On peut également choisir p égal au nombre de blocs nuls au sein du secteur. Ce choix engendre un coût matériel important : il nécessite de compter le nombre de blocs nuls lors de la sélection du bloc à éjecter. Cependant, l'opération d'allocation dans le cache n'est pas sur le chemin critique d'accès.

3.3.3 Performances

Dans cette section, nous allons évaluer les performances d'un *Zero-Content Augmented Cache* unifié. Les simulations sont effectuées avec Simics, un simulateur x86. Ce simulateur est détaillé dans la section 4.2.2.2 du chapitre 4. En effet, les nombreuses simulations nécessaires pour évaluer la politique de remplacement ne permettent pas une évaluation dans un temps réaliste avec Sesc.

3.3.3.1 Taille des secteurs

L'une des premières limitations de notre évaluation est la taille des pages du système. Elle est de 4 Ko pour la majorité des pages physiques d'un système avec un processeur x86. Cependant, le cache unifié peut accueillir des secteurs atteignant jusqu'à 32 Ko.

Dans le cas d'un secteur contenant plusieurs pages physiques différentes, le comportement du cache est dégradé. Les différentes pages physiques n'ont pas nécessairement

les mêmes fréquences d'accès. Cependant, les secteurs ne nécessitent pas de *writeback*. De plus, avec des pages de 32 Ko, la taille du cache de zéros est huit fois plus grande qu'avec des pages de 4 Ko.

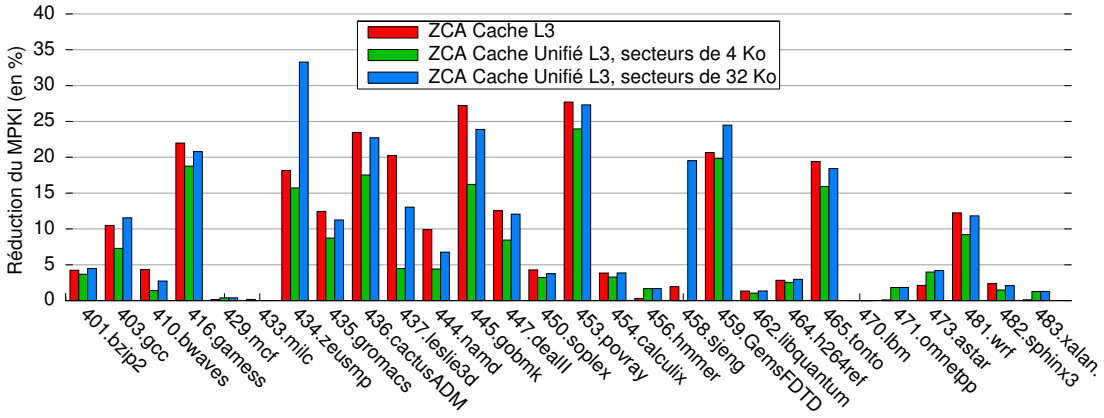


Figure 3.14 – Réduction du nombre d'échecs pour différentes tailles de secteurs.

La figure 3.14 représente la réduction du nombre d'échecs pour différents types de *Zero-Content Augmented Cache*. Le premier est un *Zero-Content Augmented Cache* standard tel que décrit dans la section 3.1. Les deux autres sont des *Zero-Content Augmented Cache* unifiés avec des secteurs de 4 Ko et de 32 Ko.

De manière générale, on ne constate pratiquement aucune augmentation du nombre d'échecs d'un *Zero-Content Augmented Cache* unifié par rapport à un cache traditionnel. Dans la configuration utilisant des pages de 4 Ko, seules les applications *458.sjeng* et *470.lbm* affichent une infime augmentation respective de $0,4\%$ et $0,003\%$. Ces valeurs sont trop faibles pour être visible sur la figure 3.14. Elles résultent de la diminution de l'associativité du cache qui passe de huit à six. Celle-ci n'est pas complètement compensée par les succès sur les secteurs.

On constate que pour majorité des applications, un *Zero-Content Augmented Cache* unifié avec des secteurs de 4 Ko se comporte moins bien qu'un *Zero-Content Augmented Cache* standard. Seules *456.hmmer*, *471.omnetpp*, *473.astar* et *483.xalancbmk* affichent un gain de quelques pourcents.

Le comportement d'un *Zero-Content Augmented Cache* unifié utilisant des secteurs de 32 Ko est bien meilleur. Pour de nombreuses applications, il est comparable à un *Zero-Content Augmented Cache* standard. Certaines applications comme *434.zeusmp* et *458.sjeng* tirent un bénéfice important de la plus grande taille de l'espace de zéros stockables.

Utiliser des secteurs de 32 Ko parait constituer le meilleur choix. Sur un système utilisant des pages de 32 Ko, les résultats seraient très certainement meilleurs [6].

6. Dans la suite des expérimentations, nous considérerons que l'OS manipule des pages de 32 Ko.

3.3.3.2 Politique de remplacement

L'utilisation d'un cache unifié permet d'envisager de nombreuses adaptations de la politique de remplacement pour tenir compte des différents coûts d'un échec sur un secteur ou sur un bloc. Dans la section 3.3.2.4, nous avons proposé plusieurs modifications de LRU, accordant un poids p plus fort aux secteurs évitant leurs éjections. Nous avons aussi proposé une politique adaptative, dans laquelle p représente le nombre de blocs nuls dans le secteur.

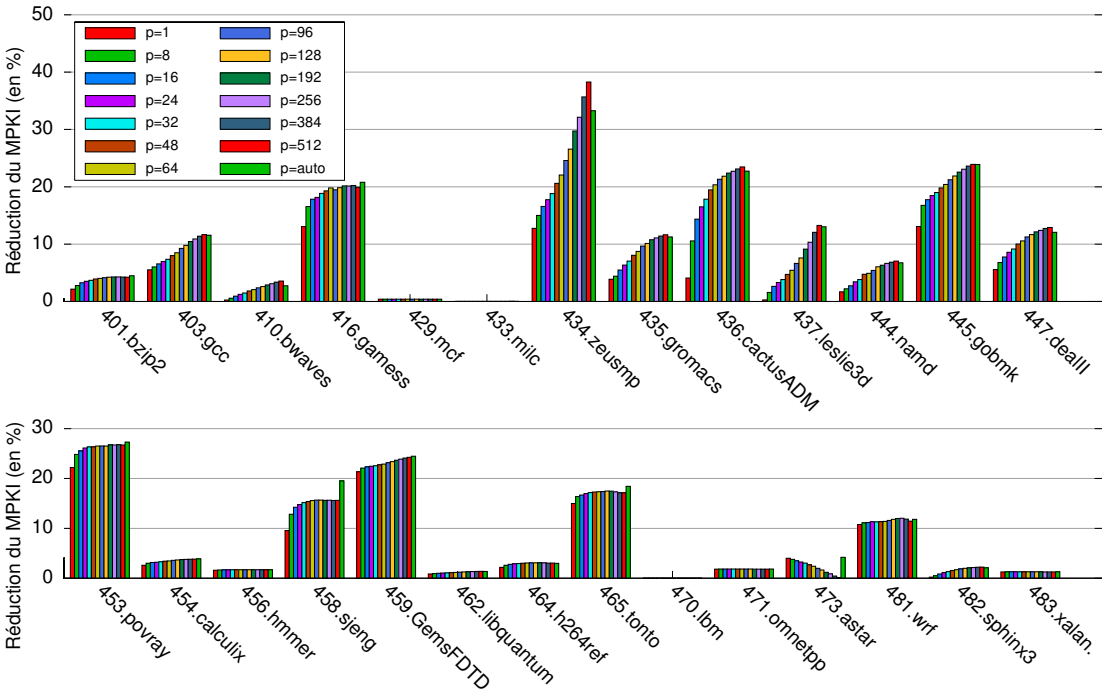


Figure 3.15 – Réduction du nombre d'échecs pour différentes politiques de remplacement. La dernière barre *p=auto* est la politique adaptative.

La figure 3.15 présente la réduction du nombre d'échecs pour différentes valeurs de p. La valeur $p = 1$ correspond à une politique LRU non-modifiée.

Pour presque toutes les applications, le taux d'échecs diminue avec l'augmentation de p. Une valeur comprise entre 192 et 512 semble être un bon choix. Seule l'application *473.astar* permet une meilleure réduction du nombre d'échecs pour de petites valeurs de p. Cela est dû à la présence de nombreux secteurs nuls qui sont très peu ré-accédés. La valeur de $p = 1$ permet d'éjecter rapidement ces secteurs pour laisser de la place pour les blocs non-nuls.

La politique adaptative est un bon compromis. Elle permet d'atteindre une bonne réduction pour toutes les applications. Pour *458.sjeng* et *465.tonto*, cette réduction est supérieure à la meilleure des politiques statiques. Cette amélioration vient de l'évolution de l'utilisation des blocs nuls au cours de la simulation.

3.4 Conclusion

Nous avons constaté au Chapitre 2 que pour de nombreuses applications, une part non-négligeable des accès se fait sur des blocs complètement nuls. Nous avons aussi constaté que ces accès ont une localité temporelle plus faible que les accès à des blocs non-nuls.

Dans ce chapitre, nous avons proposé un mécanisme permettant de stocker de façon efficace ces blocs nuls à l'extérieur du cache principal. Les blocs nuls sont regroupés en secteurs pouvant atteindre la taille d'une page. Ainsi, un bloc nul n'occupe qu'un peu plus d'un bit de stockage.

Nos évaluations ont montré l'efficacité de ce mécanisme pour réduire le nombre d'échecs du dernier niveau de cache, et donc le nombre d'accès mémoire. Un tiers des applications testées montrent une augmentation de l'IPC comprise entre 2 et 54%.

De plus, nous avons montré que le cache de zéros et le cache principal pourraient être fusionnés.

Chapitre 4

Mémoires compressées

Depuis les débuts de l'informatique, les besoins des applications en quantité de mémoire ont toujours été croissants. A chaque nouvelle génération, les systèmes sont de plus en plus gourmands, les jeux de données augmentent, rendant nécessaire l'utilisation d'une mémoire principale de plus en plus grande. Ce besoin important de mémoire est encore renforcé par l'écart important entre le temps d'accès à la mémoire principale et le temps d'accès au disque dur. Avec les technologies actuelles, l'ordre de grandeur d'un accès mémoire est d'une centaine de cycles, alors que celui du disque est d'une dizaine de millions de cycles. Il n'est en conséquence pas envisageable de s'appuyer sur le disque dur pour étendre la taille de la mémoire. Une application qui travaille sur un jeu de données qui ne tient pas en mémoire voit alors ses performances s'effondrer. Cela conduit à utiliser une mémoire principale de grande taille. Or, la mémoire représente une part importante du coût d'un ordinateur moderne. Il est par conséquent opportun de développer des techniques permettant de la compresser.

Nous avons constaté dans le chapitre 2 qu'une part importante des applications accèdent à de nombreux blocs nuls. Ces blocs nuls sont principalement présents dans les accès au dernier niveau de cache et sur le bus mémoire. Il faut cependant distinguer le taux dynamique de blocs nuls du taux statique de blocs nuls. Dans le chapitre 2, nous avons mesuré le taux de blocs nuls *dans les accès*, il s'agit d'un *taux dynamique*. A l'opposé, le taux de blocs nuls réellement *présents en mémoire* est un *taux statique*. La compression de la mémoire va dépendre du taux statique de blocs nuls. Un fort taux dynamique permet d'espérer un fort taux statique, mais n'en est pas une garantie. Comme nous l'avons présenté dans le chapitre 1, de nombreuses études ont montré que le contenu de la mémoire est fortement compressible au moyen d'algorithmes de compression assez lourds [90, 56, 75, 12, 47, 91, 13, 30, 48, 29, 96, 102], mais aussi en utilisant des algorithmes très simple [34, 74] comme FPC [8].

Dans notre publication [32], nous avons proposé une architecture matérielle de mémoire compressée nommée *Decoupled Zero Compressed Memory*. Elle est capable d'utiliser la présence de blocs nuls dans la mémoire principale. Comme toutes les propositions

précédentes de mémoire compressée matériellement [90, 34, 56], elle permet d'exploiter plus efficacement l'espace mémoire physique afin de faire fonctionner les mêmes applications dans une mémoire de plus petite taille. Ainsi, la quantité de mémoire peut être réduite. Notre proposition de mémoire compressée permet aussi de réduire la latence d'accès grâce à un mécanisme proche du *Zero-Content Augmented Cache*.

Ce chapitre est organisé en trois parties. Dans une première partie, nous allons analyser le taux de blocs nuls ou compressibles statiques présents en mémoire. Dans une deuxième partie, nous présenterons l'architecture de la *Decoupled Zero Compressed Memory* et analyser en détail son comportement. Pour finir, dans une troisième partie, nous proposerons une extension de la *Decoupled Zero Compressed Memory* capable d'utiliser l'algorithme de compression FPC.

4.1 Taux de blocs compressibles en mémoire

Comme nous l'avons présenté au chapitre 1, de nombreuses études ont montré que les contenus du cache [59, 92, 101, 103, 100, 24, 7, 8, 48, 70, 71, 89] et de la mémoire [90, 56, 75, 12, 47, 91, 13, 30, 48, 29, 96, 102, 34, 74] sont souvent fortement compressibles. Ces études ont été détaillées dans le chapitre 1. *Ekman et Stenström* [34] ont remarqué que de nombreuses applications manipulent une grande quantité de blocs nuls. Ces blocs sont souvent des blocs de 64 octets complètement nuls. Pour les SPEC CPU 2000, ils ont mesuré que 30% des blocs de 64 octets présents en mémoire était nuls. Pour *gcc*, ce taux monte à 80% de blocs nuls. Dans le chapitre 2, nous avons nous aussi mesuré un fort taux de blocs nuls dans les accès. Il s'agissait alors d'un taux dynamique de blocs nuls. Ici, nous allons mesurer le taux statique, c'est-à-dire le nombre de blocs nuls réellement présents en mémoire dans les pages accédées.

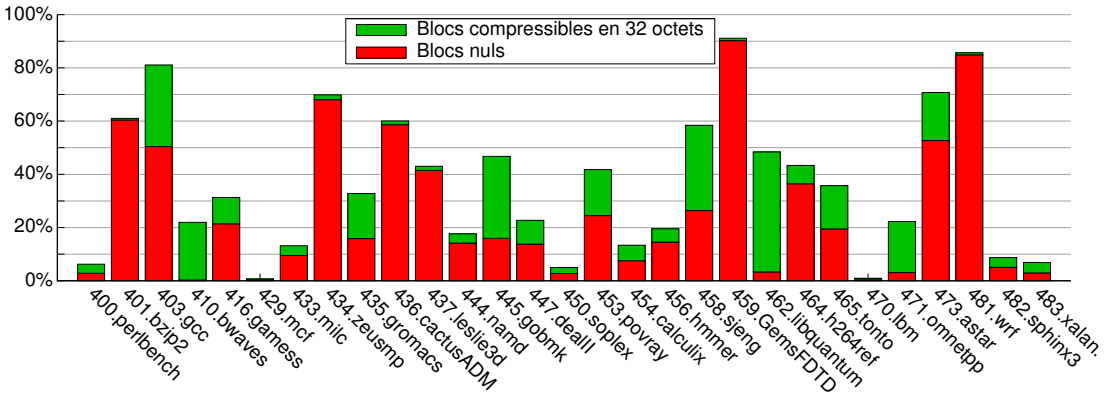


Figure 4.1 – Taux de blocs de 64 octets nuls ou compressibles en 32 octets avec FPC après 50.10^9 instructions présents en mémoire dans les pages accédées.

Nos propres simulations confirment les résultats d'*Ekman et Stenström*. Nous avons utilisé l'infrastructure de simulation Simics de Virtutech. Nous prenons donc en compte tout le contenu de la mémoire, y compris la partie utilisée par le système d'exploitation, ici Linux 2.6. Nous avons réalisé une capture du contenu de la mémoire après avoir simulé cinquante milliards d'instructions et gardé les pages accédées par l'application. Plus de détails sur la méthode expérimentale sont donnés dans la partie 4.2.2.2 page 91.

La figure 4.1 page précédente représente la proportion de blocs de 64 octets nuls ou compressibles par l'algorithme FPC en 32 octets présents en mémoire pour les applications des SPEC CPU 2006. Les mesures effectuées sur les SPEC CPU 2000 sont très semblables.

4.1.1 Taux de blocs nuls

On pourrait penser qu'un algorithme de compression ne ciblant que les blocs nuls ne compresserait qu'une partie marginale de la mémoire. Or, l'histogramme 4.1 confirme que de nombreuses applications ont un taux de blocs nuls présents en mémoire assez élevé. Il dépasse même 85% pour *459.GemsFDTD* et *481.wrf*. Utiliser une mémoire compressée matériellement ciblant ces blocs nuls permet d'envisager de limiter l'occupation mémoire de certaines applications, d'autant plus que les blocs nuls sont très fortement compressibles : ils peuvent être représentés avec un seul bit.

4.1.2 Taux de blocs compressibles avec FPC

Avant compression		Après compression		
Motif	Valeur	Préfixe	Valeur	Taille
mot nul	`00000000`	00	–	2
mot de 8 bits	`000000ab`	01	`ab`	10
mot de 16 bits signé	`SSSSabcd`	10	`abcd`	18
mot non-compressé	`abcdefgh`	11	`abcdefgh`	34

Table 4.1 – FPC à quatre motifs (S est l'extension du bit de signe)

Nous avons aussi mesuré le taux de blocs compressibles en utilisant FPC [29]. Cet algorithme est appliqué à chaque mot de quatre octets. Nous utilisons quatre motifs : entier nul, entier non-signé de 8 bits, entier de 16 bits avec extension du bit de signe, et entier 32 bits (non-compressé). Un bloc compressé avec FPC occupe donc une taille comprise entre 32 et 544 bits (soit 4 à 68 octets). Afin de réduire l'espace occupé par les

mots nuls en fin de blocs, les derniers bits nuls ne sont pas stockés. Lors de la décompression, le bloc compressé est complété d'autant de zéros que nécessaire. Ainsi, un bloc totalement nul n'occupe plus aucun bit une fois compressé. Un bloc non-nul compressé a alors une taille comprise entre 3 [1] et 544 bits.

Ici, nous ne considérons que trois cas : soit le bloc de 64 octets est complètement nul, soit il est compressible en au plus 32 octets avec FPC, soit il est non compressé. Distinguer d'autres tailles de blocs compressés engendrerait un surcoût matériel trop important. L'histogramme 4.1 montre que la plupart des blocs compressibles sont des blocs nuls. Cependant, pour certaines applications, la part de blocs compressibles est importante. Par exemple, *403.gcc*, *410.bwaves*, *435.gromacs*, *445.gobmk*, *453.povray*, *458.sjeng*, *462.libquantum*, *465.tonto*, *471.omnetpp* et *473.astar* utilisent plus de 17% de blocs compressibles. Ainsi, une mémoire compressée utilisant FPC peut potentiellement apporter un gain par rapport à une compression ne considérant que les blocs nuls.

4.2 Decoupled Zero-Compressed Memory

Comme nous venons de le constater, la proportion de blocs nuls présents en mémoire est assez importante pour de nombreuses applications. Dans cette section, nous présentons notre proposition [32] de mémoire compressée, la *Decoupled Zero-Compressed Memory*. Il s'agit d'une mémoire compressée matériellement qui exploite la présence de blocs nuls. Comme *Ekman et Stenström* [34], notre contribution vise des blocs mémoire de la granularité d'une ligne de cache, soit ici 64 octets.

4.2.1 Architecture

L'architecture globale est représentée dans le schéma 4.2 page ci-contre. Elle se traduit par l'ajout entre le dernier niveau de cache et la mémoire principale d'un contrôleur de mémoire compressée. Ce contrôleur est capable de traduire rapidement les adresses de l'espace physique non-compressé en adresse dans l'espace physique compressé. Pour cela, il utilise un petit cache, comparable au TLB, qui garde une partie des structures de contrôle. Cette architecture est conforme aux principes que nous avons définis dans la section 1.4.2.4 page 34. Elle peut être adaptée au système à mémoire distribuée en utilisant un contrôleur de mémoire compressée par mémoire.

Notre proposition de mémoire compressée s'appuie principalement sur des idées issues du *Decoupled Sectored Cache* [80] et du *Zero-Content Augmented Cache* [31] présentés dans le chapitre précédent.

1. Les blocs dont seul le premier mot est non-nul et vaut 0x80 ou 0xFFFF8000 se compressent en respectivement 011 et 101.

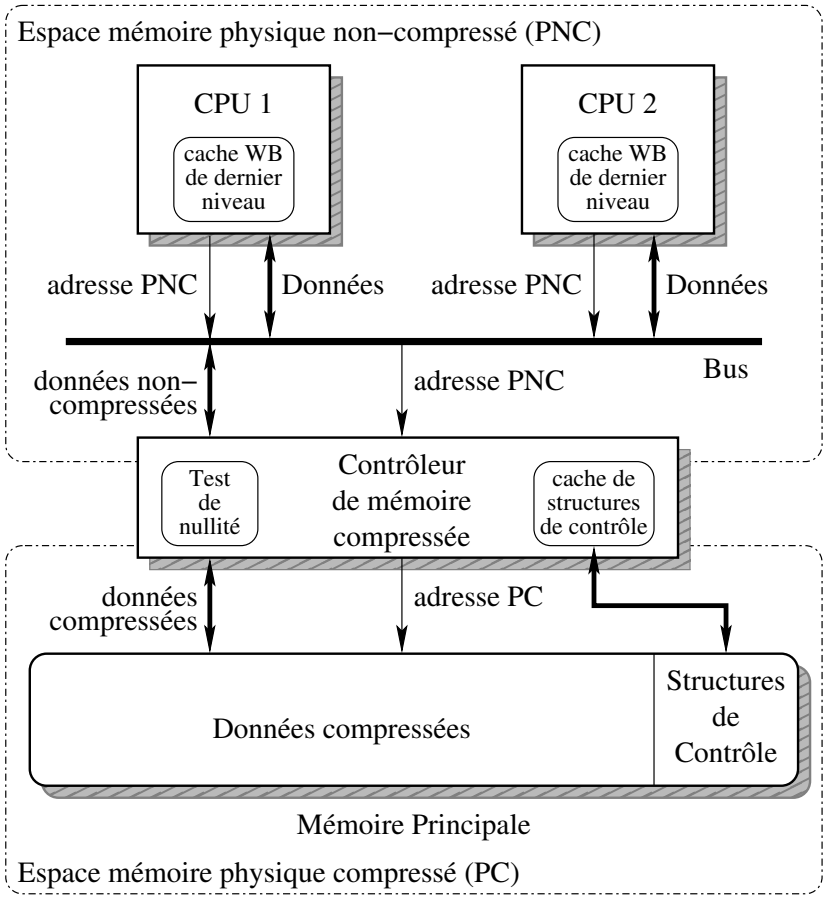


Figure 4.2 – Architecture matérielle de la *Decoupled Zero-Compressed Memory*

4.2.1.1 Structure

Afin de pouvoir gérer la mémoire compressée avec des structures de contrôle de tailles raisonnables, nous avons divisé la mémoire principale en régions de tailles égales que nous appellerons espaces physiques compressés (espace PC). La taille S de ces espaces PC considérés dans cette étude est comprise entre 128 et 1024 fois la taille P d'une page physique non compressée (page PNC), c'est-à-dire de 512 Ko à 4 Mo pour des pages de 4 Ko.

Chaque page physique est allouée dans un espace PC donné, c'est-à-dire que ses blocs non-nuls sont tous représentés au sein du même espace PC. Les blocs nuls ne sont pas stockés dans l'espace mais sont représentés par un bit dans les structures de contrôle. Pour stocker les blocs non-nuls, la mémoire est gérée comme un *Set-associative*

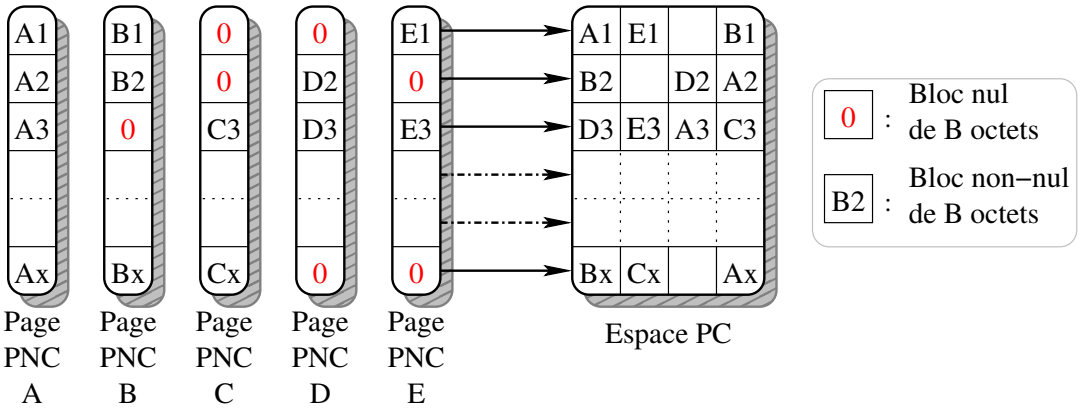


Figure 4.3 – 5 pages Physiques Non-Compressées (pages PNC) sont affectées à un espace Physique Compressé (espace PC) d'une taille de 4 pages. Chaque bloc non-nul est affecté à l'une des 4 lignes mémoire possibles.

decoupled sectored cache [80]. Ici, une page est traitée comme un secteur dont tous les blocs ne sont pas forcement présents. Chaque page est allouée dans un espace. De ce fait, chaque bloc non-nul de la page PNC a S/P positions possibles dans un espace PC. L'allocation est représentée sur le schéma 4.3. Cinq pages sont stockées dans un espace PC. L'espace est découpé en *sets* de quatre lignes. Le premier *set* contient dans cet exemple les blocs A1, E1 et B1. Le troisième *set* (D3, E3, A3, C3) illustre un *set* saturé.

Accès en lecture

La lecture d'un bloc dans la mémoire compressée suit le mécanisme de traduction d'adresse décrit sur la figure 4.4 page suivante. L'adresse physique non-compressée est découpée en trois champs AP, P_{offset} et U selon le format $AP*P+P_{offset}*B+U$, avec P la taille d'une page et B la taille d'un bloc. Chaque page physique non-compressée possède un descripteur de page associé, dont l'index est AP. Ce descripteur de page contient un pointeur AE sur l'espace dans lequel la page est stockée. Il contient aussi pour chaque bloc de la page un bit N représentant si le bloc est nul ou non et un pointeur de voie V pour adresser le bloc dans le *set* lorsqu'il est non-nul.

Lors d'une lecture, le descripteur de page est accédé. Si le bit N correspondant au bloc demandé est mis à vrai, le bloc est nul. Aucun accès à l'espace PC n'est alors nécessaire. S'il est mis à faux, les champs P_{offset} et U de l'adresse PNC ainsi que AE et V contenu dans le descripteur de page sont utilisés. Ils permettent de générer l'adresse PC : $AE * S + P_{offset} * S/P * B + V * B + U$, avec S la taille d'un espace PC. Pour les constantes S, P et B, le choix de puissances de deux permet une traduction d'adresse sans aucun calcul arithmétique.

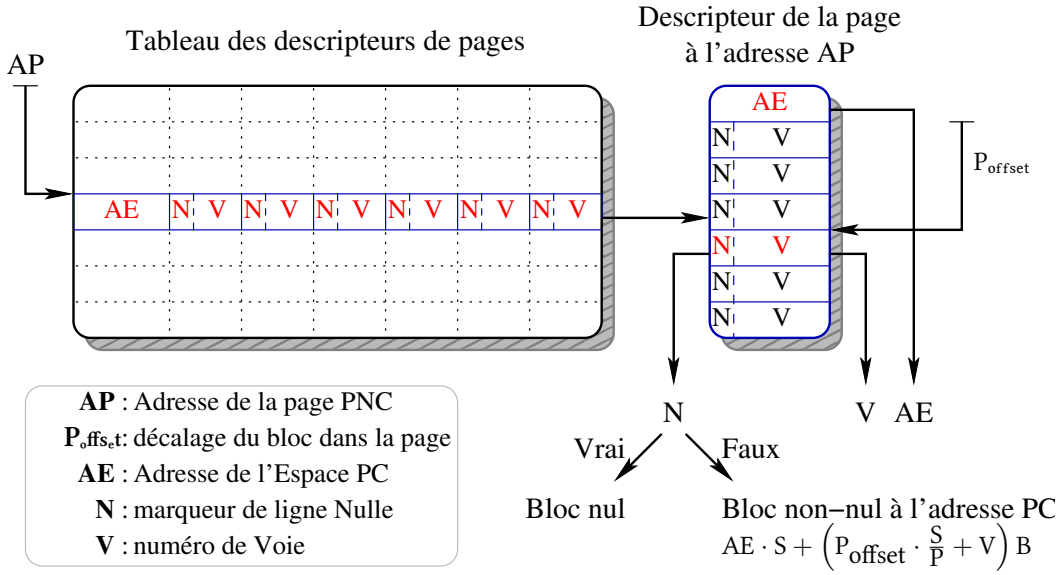


Figure 4.4 – Traduction d'adresse pour la lecture d'un bloc en mémoire compressée. Le descripteur de page PNC à l'adresse AP est lu. Il contient l'adresse de l'espace physique compressé AE où est stockée la page et, pour chaque bloc, un marqueur de bloc nul N et un pointeur de voie V.

Accès en écriture

Lors de l'écriture d'un bloc non compressé, quatre cas peuvent apparaître selon la nullité du bloc avant et après l'écriture :

- *écriture d'un bloc nul à la place d'un bloc nul :* aucune action. Le descripteur de page contenait déjà le bit N positionné.
- *écriture d'un bloc non-nul à la place d'un bloc non-nul :* l'adresse PC du bloc est obtenue de la même manière que pour la lecture. Le bloc est ensuite mis à jour en mémoire compressée avec sa nouvelle valeur.
- *écriture d'un bloc nul à la place d'un bloc non-nul :* le bit N est mis à vrai pour signaler un bloc nul. Ensuite, la ligne de l'espace PC précédemment occupée par le bloc est libérée. Ce mécanisme est décrit dans le paragraphe suivant.
- *écriture d'un bloc non-nul à la place d'un bloc nul :* une ligne de l'espace PC au sein du set correspondant à ce bloc est allouée. Le pointeur de voie est ensuite mis à jour.

Pour gérer ces deux derniers cas, un descripteur (figure 4.5 page suivante) est associé à chaque espace PC. Il est principalement constitué de bits de validité v (un par ligne de l'espace PC). Ces bits constituent un *bitmap* indiquant pour chaque ligne si elle est libre ou non.

Pour libérer une ligne dans un espace PC, il faut accéder à l'espace PC d'index AE et mettre le bit v correspondant à faux.

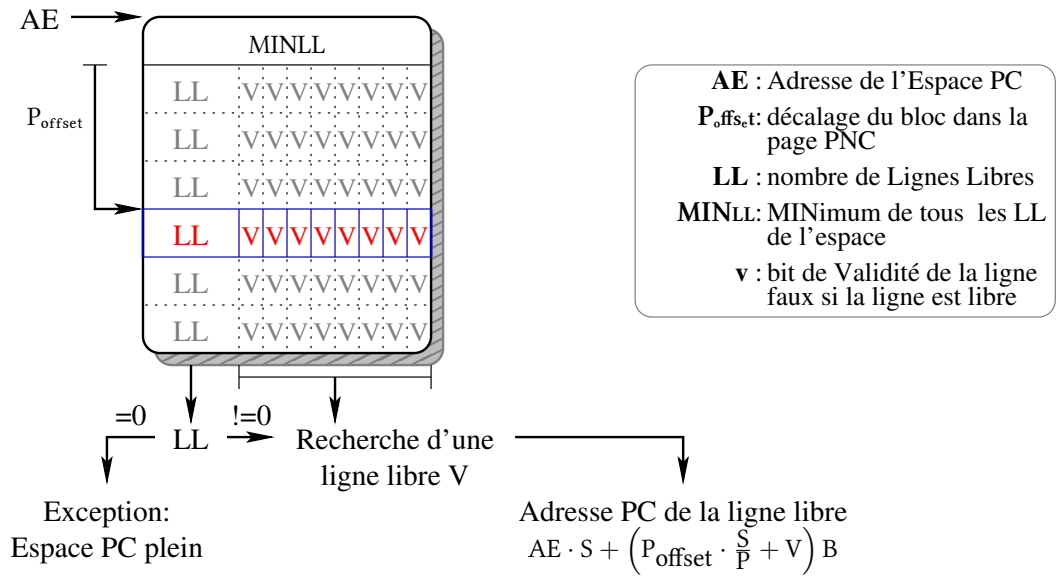


FIGURE 4.5 – Descripteur d'espace PC. Recherche d'une ligne libre.

Pour allouer une ligne dans un espace PC, il faut trouver une ligne libre dans le *set* correspondant. La figure 4.5 décrit ce mécanisme ; les bits v sont examinés dans le but d'en trouver un à faux.

Dans le cas rare d'une écriture nécessitant une allocation (nul vers non-nul) et qu'aucune ligne n'est libre pour l'allocation, une exception est levée. La page PNC entière est alors déplacée vers un autre espace PC avec au moins une ligne libre par set. Ce déplacement se fait de mémoire vers mémoire. En cas de saturation de la mémoire, aucun espace avec une ligne libre par set n'est trouvé. Il faut alors libérer de l'espace en mémoire principale en éjectant une ou plusieurs pages (*swap-out*).

Afin de gérer le déplacement et l'allocation des pages dans les espaces PC, les bits de validité (v) suffisent. Cependant, stocker des informations redondantes permet de simplifier le mécanisme de recherche d'espace libre. Des compteurs LL représentant le nombre de Lignes Libres pour chaque *set*, ainsi qu'un compteur global à l'espace MIN_{LL} défini comme le MINimum de tous les compteurs LL, sont ajoutés au descripteur d'espace PC. Lors d'une écriture provoquant le changement d'état d'une ligne, dans le cas général, seuls deux compteurs sont mis à jour : le LL correspondant au *set*, et si besoin, le MIN_{LL} de l'espace.

Le cas exceptionnel d'une écriture changeant un bloc nul en bloc non-nul sans aucune ligne de libre dans le *set* peut être rapidement détecté grâce à $LL = 0$. La page est alors déplacée vers un autre espace PC. Le nouvel espace PC est choisi grâce au compteur MIN_{LL}. S'il est non-nul, l'espace contient de façon certaine au moins assez de place pour stocker la page quel que soit le nombre de blocs non-nuls qu'elle contient.

4.2.1.2 Coût de stockage des structures de contrôle

Les structures de contrôle nécessaires sont les descripteurs de pages et les descripteurs d'espace, respectivement associés aux pages PNC et aux espaces PC.

Le descripteur de page (figure 4.4 page 85) contient un pointeur d'espace PC, et, pour chaque bloc de la page, un pointeur de voie et un bit de nullité. Si l'on choisit comme dans notre méthode expérimentale des pages et des espaces PC de tailles respectivement $P = 4$ Ko et $S = 4$ Mo, le pointeur de voie sera de $log_2(\frac{S}{P}) = 10$ bits. Avec des blocs de taille $B = 64$ octets, un descripteur de page contient $\frac{P}{B} = 64$ pointeurs de voie. Un pointeur d'espace de 32 bits permet d'adresser une mémoire compressée de 2^{54} octets. Cela représente une taille de descripteur de page de : $\frac{P}{B} \cdot (log_2(\frac{S}{P}) + 1) + 32 = 736$ bits, soit 92 octets.

Le descripteur d'espace (figure 4.5 page ci-contre) contient un bit v de validité par ligne, soit $\frac{S}{B} = 65536$ bits par espace PC. Les compteurs LL et MIN_{LL} sont alors sur $\lfloor log_2(\frac{S}{P}+1) \rfloor = 11$ bits. Cela représente au total $\frac{S}{B}+(\frac{P}{B}+1)\lfloor log_2(\frac{S}{P}+1) \rfloor = 66251$ bits, soit 8282 octets.

L'espace total requis par les structures de contrôle dépend de la taille de l'espace non-compressé projeté sur la mémoire compressée. En prenant la configuration précédente avec un facteur moyen de compression de $1,5$, cela représente $1.5 * 1024 * 92 + 8282 = 149594$ octets par espace PC soit $3,6\%$ de la mémoire. Pour une grande mémoire compressée, ces structures doivent être stockées en mémoire principale (schéma 4.2 page 83).

4.2.1.3 Contrôleur de mémoire compressée

Dans tout système, le contrôleur mémoire gère les accès à la mémoire principale. Pour la *Decoupled Zero-Compressed Memory*, le contrôleur de mémoire compressée peut être intégré au sein du contrôleur mémoire.

Pour chaque accès, le contrôleur de mémoire compressée doit déterminer si un bloc est nul ou non. S'il est non-nul, il doit traduire l'adresse PNC en adresse PC. Lors d'une écriture changeant l'état d'un bloc, le contrôleur de mémoire compressée doit allouer ou libérer une ligne. Lors de l'allocation, si aucune ligne n'est libre dans l'ensemble des lignes possibles pour le bloc donné, il doit déplacer la page vers un autre espace PC. Si cette tâche reste relativement exceptionnelle, il peut être aidé par le système d'exploitation.

Toutes ces tâches nécessitent d'avoir accès aux structures de contrôle de la mémoire compressée. La traduction d'adresse PNC en adresse PC est très fréquente. Elle est nécessaire pour tout accès à un bloc non-nul. De plus, elle est sur le chemin d'accès critique. À l'opposé, le déplacement de pages et l'allocation de lignes sont des événements beaucoup plus rares. Ils ne se produisent que pour certaines écritures. L'impact de cette latence sur les performances est donc beaucoup plus faible.

Pour permettre une traduction d'adresse rapide entre les espaces d'adressage PNC et PC, les descripteurs de pages doivent être mis en cache dans le contrôleur de mémoire compressée, comme illustré sur la figure 4.2. Ce cache de descripteurs de pages peut immédiatement répondre aux lectures de blocs nuls sans accéder à la mémoire principale. Dans ce cas, le temps d'accès est diminué. Ce cache se comporte comme le *Zero Content Augmented Cache* décrit au chapitre 3.

4.2.1.4 Amélioration de la distribution des blocs nuls

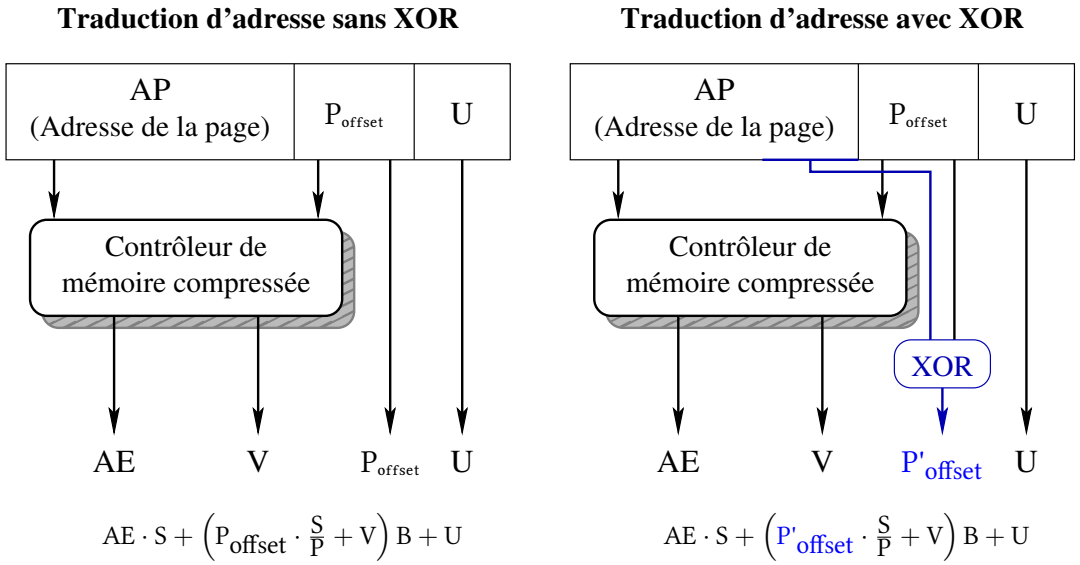


Figure 4.6 – Fonctionnement du XOR. Lors de la traduction d'adresse, le numéro du *set* de l'espace où est stockée la page est obtenu après un xor avec une partie de l'adresse de la page.

La *Decoupled Zero-Compressed Memory* a un comportement de cache associatif par *set* dans lequel les blocs nuls ne coûtent pas de ligne. Malheureusement, les résultats expérimentaux ont montré que les blocs nuls ne sont pas distribués de façon homogène au sein des pages. Les blocs nuls sont plus souvent présents à la fin des pages qu'au début. Avec une projection directe des blocs dans les *sets* selon leur adresse, les *sets* correspondants aux blocs de début de page seront plus souvent saturés.

Pour rendre la distribution des blocs nuls plus homogène, nous utiliserons un ou-exclusif pour calculer l'index du *set*. Au lieu d'utiliser le décalage du bloc dans la page, P_{offset}, on utilise le résultat du ou-exclusif entre P_{offset} et les bits de poids faible de l'adresse de la page. Ceci est illustré dans la figure 4.6.

4.2.1.5 Amélioration par rapport aux propositions antérieures

Par rapport à la compression logicielle

Lors de l'utilisation d'une mémoire avec compression logicielle, seule une partie de la mémoire est compressée. Elle est appelée cache compressé. Lors d'un accès à une donnée dans ce cache compressé, le système d'exploitation recopie et décompresse la page entière dans la partie non-compressée. La taille du cache compressé a donc une grande importance. S'il est trop grand, trop de recopies de pages vont être nécessaires. S'il est trop petit, la compression est inefficace. En effet, la loi d'Amdhal rappelle que les gains possibles sont bornés par la taille de la fraction optimisée. Ainsi, si 10% de la mémoire est compressée avec un taux de compression de $0,7$, le gain d'espace n'est que de 7%. Le taux de compression doit impérativement être élevé pour exploiter l'espace occupé par le cache compressé en mémoire principale.

Notre proposition utilise une mémoire compressée matériellement. La mémoire est entièrement compressée, les accès se faisant sans recopie de la page. Une grande zone étant compressée, la contrainte d'efficacité de l'algorithme de compression est beaucoup moins forte. Nous pouvons donc utiliser un algorithme moins efficace, mais plus rapide et avec une granularité plus faible. La latence d'accès aux données compressées est très largement inférieure.

Par rapport à la technologie IBM MXT

La technologie IBM MXT, décrite dans la partie 1.4.2.1 page 26, utilise des blocs de 1 Ko. La *Decoupled Zero-Compressed Memory* se distingue principalement par l'utilisation de blocs de 64 octets, et d'un algorithme de compression simple : nul ou non-nul.

L'impact du temps de décompression sur les performances est important dans l'architecture IBM MXT. Un cache de 32 Mo permet de diminuer le nombre de décompressions pour les applications dont le *working-set* tient dans ce cache. Nous proposons une technique permettant une décompression presque instantanée. Il n'est alors plus nécessaire de garder ce cache.

Notre proposition permet aussi de réduire la granularité des accès à la mémoire principale. Il en résulte un temps d'accès plus court aux données lors d'un défaut de cache, et surtout une utilisation moindre de la bande passante mémoire.

Par rapport à la proposition d'*Ekman et Stenström*

Comme nous l'avons déjà évoqué dans la partie 1.4.2.2 page 29, l'approche d'*Ekman et Stenström* [34] pose problème lors de l'écriture de blocs dont la taille augmente. Cette approche nécessite en effet de déplacer fréquemment toutes les données suivantes dans la sous-page et dans la page. Dans certains cas, la page doit même être déplacée ailleurs.

Notre mémoire compressée rend de tels déplacements beaucoup plus exceptionnels. Les espaces libres dus à la fragmentation interne sont mutualisés entre les blocs de dif-

férentes pages au sein d'un *set*. Dans le cas rare du déplacement d'une page, toutes les pages de l'espace PC peuvent bénéficier de l'espace ainsi libéré.

Grâce à l'utilisation d'un contrôleur mémoire, notre architecture permet aussi de traduire l'adresse lors d'un *writeback* du dernier niveau de cache et est utilisable sur des architectures multi-cœurs, ou multiprocesseurs.

4.2.2 Évaluation des performances

4.2.2.1 Métriques choisies

L'évaluation des performances doit permettre de mesurer deux objectifs distincts. Dans un premier temps, nous devons évaluer dans quelle proportion l'utilisation de la compression augmente la taille de la mémoire principale. Dans un second temps, nous devons évaluer l'impact sur les performances lorsque l'application tient en mémoire, puisque le système peut aussi être utilisé avec ce type d'application.

Défauts et déplacements de pages

L'intérêt principal venant de l'utilisation d'une mémoire compressée est d'augmenter la taille du *working-set* que le système peut gérer sans avoir recours au *swap* sur un périphérique de stockage.

Le temps d'accès à un disque dur est de l'ordre de 10 ms, soit environ 30.10^6 cycles pour un processeur moderne. De ce fait, même un très faible nombre de défauts de pages, tel que cent pour un milliard d'instructions (soit 3.10^9 cycles), n'est pas acceptable. Du fait de son très fort impact sur les performances de l'application, nous avons choisi le taux de défauts de pages par milliard d'instructions comme métrique. Nous allons le mesurer sur un large spectre de tailles de mémoires pour évaluer l'augmentation de l'espace mémoire disponible pour l'application grâce à la compression.

Cependant, notre mémoire compressée peut souffrir d'une saturation des espaces PC sur les écritures (cf. section 4.2.1). Une telle saturation oblige à déplacer la page vers un autre espace PC, ce qui engendre un impact sur les performances. Comme *Ekman et Stenström* [34] l'ont noté, ces déplacements sont locaux à la mémoire et peuvent être effectués en tâche de fond. Ils ont peu de chance de modifier les performances s'ils restent suffisamment exceptionnels. S'ils sont en nombre trop important, on peut envisager de modifier la politique d'allocation. On peut, par exemple, choisir un nombre X, et ne pas allouer de page dans un espace avec un $MIN_{LL} \leq X$. Mais cela augmenterait le nombre de défauts de pages, or ceux-ci sont largement plus coûteux que les déplacement de pages.

Comme seconde métrique de performances, nous présenterons donc le nombre de déplacements pour nous assurer qu'ils sont suffisamment rares quand le taux de défauts de pages permet d'envisager une exécution dans un temps réaliste.

Temps moyen d'accès mémoire

Lorsque le *working-set* d'une application tient en mémoire, utiliser une mémoire compressée ne se traduit pas en gain de performances. Généralement, il en résulte même une perte de performances due aux latences de traduction d'adresse PNC en adresse PC et de décompression du bloc.

Dans notre proposition, aucun mécanisme complexe de décompression n'est nécessaire. De ce fait, l'augmentation du temps d'accès ne peut venir que de la traduction d'adresse PNC en adresse PC. Elle est réalisée grâce au cache des descripteurs de pages.

Si le descripteur correspondant à la page PNC demandée est présent dans le cache, la traduction d'adresse ne rajoute que quelques cycles à la latence totale du contrôleur mémoire. Cette traduction pourrait même être réalisée en parallèle avec d'autres tâches au sein du contrôleur mémoire. Comme *Ekman et Stenström* l'ont constaté [34], les deux ou trois cycles supplémentaires lors d'un accès mémoire n'ont qu'un impact très limité sur les performances globales du système.

Dans le cas contraire, si le descripteur correspondant à la page PNC n'est pas présent dans le cache des descripteurs de pages, il doit être lu en mémoire. Il en résulte une augmentation significative du temps d'accès à la donnée.

Cependant, lors de la lecture d'un bloc nul avec un succès dans le cache de descripteurs de pages, le bloc nul est retourné directement au processeur sans qu'un accès à la mémoire ne soit nécessaire. Il en résulte une diminution significative du temps d'accès à la donnée.

Pour refléter ces différentes contributions antagonistes à la latence mémoire, nous allons mesurer l'augmentation et/ou la diminution du temps moyen d'accès à la mémoire. Afin de mieux caractériser l'impact sur les performances, nous allons mesurer la contribution en cycles par instruction de la latence mémoire.

4.2.2.2 Méthode expérimentale

Environnement de simulation

Nous avons choisi d'utiliser Simics pour simuler notre proposition. Il s'agit d'un simulateur permettant de faire fonctionner un système complet et d'analyser les transferts entre la mémoire et le processeur. Le simulateur Sesc que nous avons utilisé au chapitre précédent ne permet pas de simuler efficacement une mémoire compressée. En effet, aucun disque dur n'est simulé. Son évaluation nécessiterait de prendre en compte toutes les E/S, le système de fichiers, etc. Définir une latence constante pour échanger une page entre le disque et la mémoire offrirait une très mauvaise évaluation de l'IPC.

Nous avons donc choisi comme métrique le nombre de défauts de pages mémoire, c'est-à-dire le nombre d'échanges de pages entre le disque dur et la mémoire principale (*swap-in*). Ce nombre est un bon indicateur car, s'il est trop important, les performances s'effondrent.

Afin d'évaluer le taux réel de blocs nuls présents en mémoire, nous utilisons les adresses physiques. Utiliser des adresses logiques peut conduire à une mauvaise évaluation. Par exemple, la fonction *calloc()* dans la glibc (et donc eglibc) utilise un mécanisme de *copy-on-write*. Une seule page physique est mise à 0, et toutes les pages logiques pointent alors sur cette unique page physique. Utiliser des adresses logiques provoque alors une surévaluation du nombre de bloc nuls.

Configuration

CPU	processeur x86
Cache L1	32 Ko, lignes de 64 octets, 4 voies, politique LRU, write allocate
Cache L2	256 Ko, lignes de 64 octets, 4 voies, politique LRU, write allocate
Cache L3	1 Mo, lignes de 64 octets, 8 voies, politique LRU, write allocate
O.S.	Linux Red Hat - Linux kernel 2.6
Benchmarks	SPEC CPU 2000 - jeux de données ref SPEC CPU 2006 - jeux de données ref
Mémoire compressée	Espace PC de 4 Mo, pages de 4 Ko, lignes de 64 octets
Temps d'accès mémoire	250 cycles ($25 + 200 + 25$)

Table 4.2 – Configuration de base de l'environnement de simulation.

La configuration de Simics et de la hiérarchie mémoire est illustrée dans le tableau 4.2. Différentes tailles ont été simulées pour les mémoires compressées et non-compressées allant de 8 Mo à 1 Go (1280 Mo pour *434.zeusmp*). Les simulations sont effectuées sur les cinquante premiers milliards d'instructions des applications des CPU SPEC 2000 et 2006.

Afin de réaliser les simulations dans un temps raisonnable, nous avons enregistré une image instantanée de la mémoire au début de l'application ainsi que la trace de tous les accès avec leurs valeurs. Le nombre d'accès est en moyenne de 80.10^9, 1.10^9, 600.10^6 et 400.10^6 pour respectivement les caches de niveau 1, 2, 3 et la mémoire. Il est impossible de stocker 80.10^9 entrées de 80 octets. Nous avons donc réalisé la trace au niveau de la sortie du L2. Les traces produites occupent entre 1 et 50 Go par application.

Politique de remplacement de pages

Lors d'un défaut de page, la page doit être ramenée en mémoire. Si la mémoire principale ne dispose pas de suffisamment d'espace libre, i.e. tous les espaces ont un $MIN_{LL} = 0$, une ou plusieurs pages doivent être éjectées.

Pour la mémoire non compressée, nous avons choisi une politique LRU. Pour la mémoire compressée, la page doit être allouée au sein d'un espace PC. Nous avons choisi une politique dérivant de LRU. Dans une première étape, nous recherchons un espace PC capable de stocker la page ($MIN_{LL} \geq 1$). Si aucun espace ne satisfait ces conditions, la page la moins récemment utilisée est éjectée (LRU). Cependant, cela ne garantit pas qu'il y ait assez de place pour stocker la nouvelle page. Si la page éjectée contenait des blocs nuls, certains *sets* de l'espace PC peuvent rester saturés. Dans ce cas, d'autres pages de cet espace PC sont éjectées jusqu'à ce que la nouvelle page tienne de façon certaine, i.e. $MIN_{LL} \geq 1$.

Déplacement de pages lors des écritures

Comme nous l'avons précédemment mentionné, lors de l'écriture d'un bloc non-nul à la place d'un bloc nul, une saturation de l'espace PC peut apparaître. Dans ce cas, la page est déplacée vers un autre espace PC, i.e. avec $MIN_{LL} \geq 1$. Quand tous les espaces PC sont saturés, le même mécanisme de libération que décrit précédemment est appliqué.

4.2.2.3 Résultats

Nombre de défauts de pages

Les courbes de la figure 4.7 page suivante présentent le nombre de défauts de pages mémoire et de déplacements de pages pour quelques applications représentatives [2]. Le premier accès à une page n'est pas pris en compte car il ne provoque aucun accès disque. Les résultats sont représentés en nombre d'occurrences par milliard d'instructions, avec une limite à 100000. Au dessus de ce seuil, le ralentissement de l'application est tel que les performances sont inacceptables. Les points intéressants sont ceux pour lesquels le nombre de défauts de pages devient presque nul, lorsque le *working set* complet tient en mémoire.

Nous pouvons tout d'abord noter une très forte corrélation entre les applications ayant un fort taux statique de blocs nuls (figure 4.1 page 80) et celle bénéficiant de notre proposition de mémoire compressée. Elle permet de réduire la quantité de mémoire nécessaire à un fonctionnement optimal pour 16 applications sur les 26 des SPEC CPU 2000 et 14 sur les 29 des SPEC CPU 2006.

Les applications *400.perlbench*, *410.bwaves*, *429.mcf*, *450.soplex*, *454.calculix*, *462.libquantum*, *470.lbm*, *471.omnetpp* et *482.sphinx3* ne tirent aucun bénéfice de l'utilisation de la *Decoupled Zero-Compressed Memory* plutôt que d'une mémoire non-compressée. *429.mcf* et *454.calculix* sont représentées figure 4.7, les autres le sont en annexe. Ces applications ont toutes un taux statique de blocs nuls dans les pages allouées compris

2. Les autres applications des SPEC CPU 2006 sont représentées en annexe dans les figures 4.14, 4.15, 4.16 et 4.17 pages 104 à 107.

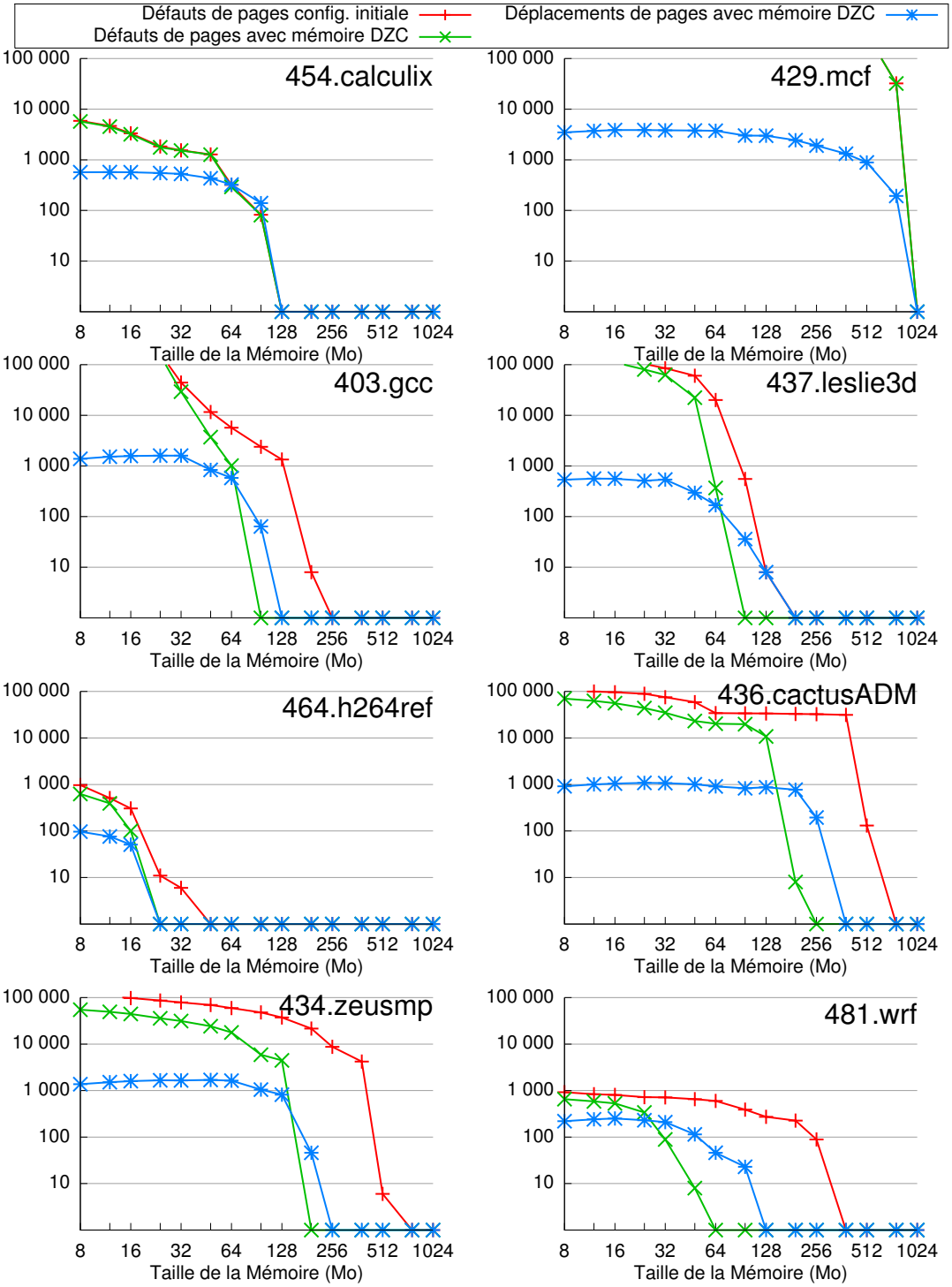


Figure 4.7 – Nombre de défauts et de déplacements de pages mémoire avec et sans *Decoupled Zero Compressed Memory* par milliard d'instructions.

entre 0 et 10% (figure 4.1). Pour que leurs *working-sets* tiennent en mémoire, la *Decoupled Zero-Compressed Memory* nécessite approximativement la même taille qu'une mémoire non-compressée.

Concernant les autres applications qui ont un taux de blocs nuls plus important, nos simulations montrent un gain à l'utilisation de la *Decoupled Zero-Compressed Memory*. Celle-ci est capable de faire fonctionner des applications dont le *working-set* est plus grand que la taille physique de la mémoire. Ce gain augmente avec le taux de blocs nuls.

C'est le cas pour les applications utilisant un taux de blocs nuls compris entre 30 et 50% telles que *403.gcc*, *437.leslie3d* et *464.h264ref*. Par exemple, pour *403.gcc*, une mémoire compressée de 96 Mo permet de contenir l'intégralité du *working-set* alors qu'une mémoire non-compressée a besoin de 256 Mo.

Pour les applications présentant un fort taux de blocs nuls, le bénéfice de la *Decoupled Zero-Compressed Memory* est plus important. Les applications *401.bzip2*, *434.zeusmp*, *436.cactusADM*, *459.GemsFDTD*,*473.astar* et *481.wrf* utilisent plus de 50% de blocs nuls. Par exemple, pour *481.wrf*, 85% des blocs sont nuls. Une mémoire compressée de 64 Mo permet de stocker le jeu de données qui nécessite 384 Mo avec un mémoire non-compressée.

Déplacements de pages

La figure 4.7 illustre aussi le nombre de déplacements de pages nécessaires. Ces déplacements ont lieu lorsqu'un bloc passe de nul à non-nul et qu'il n'y a plus de place dans le *set*. Ils se font de mémoire à mémoire. Leurs coûts sont de plusieurs ordres de grandeur moindres qu'un défaut de page.

On constate qu'ils ne sont jamais un facteur de ralentissement. Ils ne sont relativement élevés que lorsque le nombre de défauts de pages ne permet pas une exécution réaliste. Quand le *working-set* tient dans la *Decoupled Zero-Compressed Memory*, ils ne sont que quelques dizaines par milliard d'instructions, et ne risquent pas de détériorer les performances du système.

Les simulations montrent qu'il n'y a pas besoin de chercher à en diminuer le nombre. Diminuer le nombre de déplacements risque de conduire à augmenter le nombre de défauts de pages.

Temps moyen d'accès mémoire

Afin d'évaluer l'impact de la *Decoupled Zero-Compressed Memory* sur le temps d'accès moyen, nous considérons le cas où le jeu de données complet tient en mémoire. Comme nous l'avions précisé dans la section 4.2.2.1, le temps d'accès à la mémoire subit deux phénomènes antagonistes. Un échec dans le cache des descripteurs de pages peut augmenter la latence. Au contraire, un succès sur un bloc nul peut être servi directement.

Afin d'évaluer le temps moyen d'accès à la mémoire, nous avons divisé le temps d'accès en trois parties. La première partie correspond au temps d'accès du processeur au contrôleur mémoire, la deuxième partie à l'aller-retour du contrôleur mémoire à la mémoire principale, et la troisième partie au retour du contrôleur mémoire vers processeur. Lorsque l'on utilise une *Decoupled Zero-Compressed Memory*, un défaut dans le cache des descripteurs de pages provoque un aller-retour supplémentaire du contrôleur à la mémoire principale. Inversement, un succès sur un bloc nul dispense de cet aller-retour.

Dans les résultats présentés, nous considérons une latence de 25 cycles entre le processeur et le contrôleur mémoire, et une latence de 200 cycles pour un aller-retour du contrôleur à la mémoire. Ainsi, il faut 50 cycles en cas de succès sur un bloc nul, 250 cycles pour un succès sur un bloc non-nul ou pour un échec sur un bloc nul, et 450 cycles pour un échec sur un bloc non-nul.

Le tableau 4.3 page ci-contre illustre le temps d'accès relatif à la *Decoupled Zero-Compressed Memory* par rapport à une mémoire non-compressée pour des caches de descripteurs de pages allant de 11 à 736 Ko. Le temps relatif est obtenu en divisant le *Decoupled Zero-Compressed Memory* par le temps d'accès à une mémoire non-compressée. Ainsi, les valeurs inférieures à 1 traduisent un temps d'accès plus court qu'une mémoire non-compressée, grâce aux blocs nuls. Avec un cache de 11 Ko, la latence des accès mémoire de 15 applications est augmentée. L'application *473.astar* subit une augmentation de 53% de sa latence mémoire, alors que pour *481.wrf* cette latence diminue de 61%. Un cache de 184 ou 368 Ko semble un bon compromis efficacité-taille. Dans la suite des expérimentations, nous considérons un cache occupant 368 Ko.

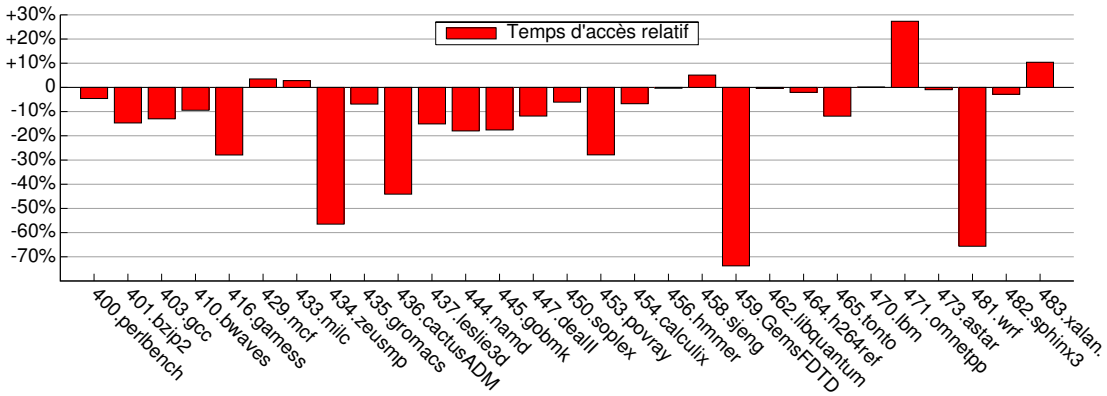


Figure 4.8 – Temps d'accès relatif à la *Decoupled Zero-Compressed Memory* par rapport à une mémoire non-compressée

La figure 4.8 représente la colonne du tableau 4.3 correspondant à un cache de 368 Ko. On constate que la latence mémoire est fortement diminuée pour la majorité des applications. Pour *459.GemsFDTD*, une diminution spectaculaire de 74% est observée. Il en

	Taille du cache de desc. de pages (en Ko)						
Application	11	23	46	92	184	368	736
400.perlbench	*1,15*	*1,12*	*1,07*	*1,03*	**0,98**	**0,95**	**0,95**
401.bzip2	*1,04*	**0,95**	**0,87**	**0,86**	**0,85**	**0,85**	**0,85**
403.gcc	**0,94**	**0,93**	**0,92**	**0,91**	**0,89**	**0,87**	**0,84**
410.bwaves	**0,91**	**0,91**	**0,91**	**0,91**	**0,91**	**0,91**	**0,91**
416.gamess	**0,80**	**0,77**	**0,75**	**0,73**	**0,72**	**0,72**	**0,72**
429.mcf	*1,22*	*1,20*	*1,15*	*1,09*	*1,05*	*1,03*	*1,02*
433.milc	*1,10*	*1,08*	*1,06*	*1,04*	*1,04*	*1,03*	*1,02*
434.zeusmp	**0,61**	**0,57**	**0,50**	**0,45**	**0,44**	**0,44**	**0,43**
435.gromacs	**0,96**	**0,95**	**0,94**	**0,94**	**0,93**	**0,93**	**0,93**
436.cactusADM	**0,60**	**0,60**	**0,59**	**0,57**	**0,56**	**0,56**	**0,56**
437.leslie3d	**0,86**	**0,86**	**0,86**	**0,85**	**0,85**	**0,85**	**0,85**
444.namd	**0,88**	**0,86**	**0,84**	**0,83**	**0,82**	**0,82**	**0,82**
445.gobmk	*1,06*	*1,02*	**0,96**	**0,88**	**0,83**	**0,82**	**0,82**
447.dealII	**0,92**	**0,91**	**0,90**	**0,89**	**0,88**	**0,88**	**0,88**
450.soplex	*1,35*	*1,31*	*1,24*	*1,14*	*1,01*	**0,94**	**0,94**
453.povray	**0,84**	**0,81**	**0,77**	**0,73**	**0,72**	**0,72**	**0,72**
454.calculix	**0,97**	**0,95**	**0,94**	**0,94**	**0,94**	**0,93**	**0,93**
456.hmmer	*1,03*	*1,01*	*1,01*	**1,00**	**1,00**	**1,00**	**1,00**
458.sjeng	*1,20*	*1,19*	*1,17*	*1,14*	*1,11*	*1,05*	**0,95**
459.GemsFDTD	**0,35**	**0,32**	**0,31**	**0,28**	**0,27**	**0,26**	**0,26**
462.libquantum	*1,00*	*1,00*	*1,00*	**1,00**	**1,00**	**1,00**	**0,99**
464.h264ref	*1,07*	*1,04*	*1,01*	**0,99**	**0,98**	**0,98**	**0,98**
465.tonto	*1,01*	**0,97**	**0,93**	**0,89**	**0,88**	**0,88**	**0,88**
470.lbm	*1,01*	*1,01*	*1,01*	*1,01*	*1,00*	*1,00*	*1,00*
471.omnetpp	*1,36*	*1,36*	*1,35*	*1,33*	*1,31*	*1,27*	*1,23*
473.astar	*1,53*	*1,47*	*1,34*	*1,16*	*1,03*	**0,99**	**0,98**
481.wrf	**0,39**	**0,37**	**0,36**	**0,35**	**0,34**	**0,34**	**0,34**
482.sphinx3	**1,00**	**0,99**	**0,99**	**0,99**	**0,99**	**0,97**	**0,97**
483.xalancbmk	*1,17*	*1,17*	*1,16*	*1,15*	*1,13*	*1,10*	*1,06*

Table 4.3 – Temps d'accès relatif à la *Decoupled Zero-Compressed Memory* par rapport à une mémoire non-compressée pour différentes tailles de cache. Les valeurs en gras représentent une configuration avec une latence plus faible

est de même pour *481.wrf*, avec une réduction de 66%. Cependant pour *429.mcf*, *433.milc* et *458.sjeng* cette latence augmente très légèrement. Pour *471.omnetpp* et *483.xalancbmk* l'augmentation est de respectivement 27 et 10%.

Toutefois, les applications n'ont pas toutes les mêmes exigences en termes de latence. Afin de mieux corréler cette latence avec la quantité d'accès à la mémoire, nous proposons de considérer le nombre moyen de cycles passés dans les accès à la mémoire par instruction, figure 4.9 page suivante. Les trois applications effectuant le plus d'accès mé-

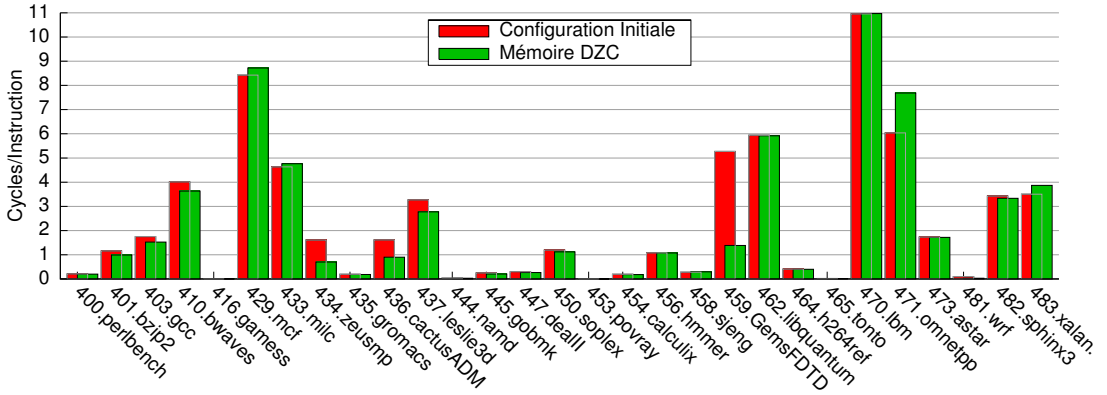


Figure 4.9 – Temps total des accès mémoires par instruction

moire sont légèrement pénalisée. *459.GemsFDTD* bénéficie fortement de la réduction de la latence.

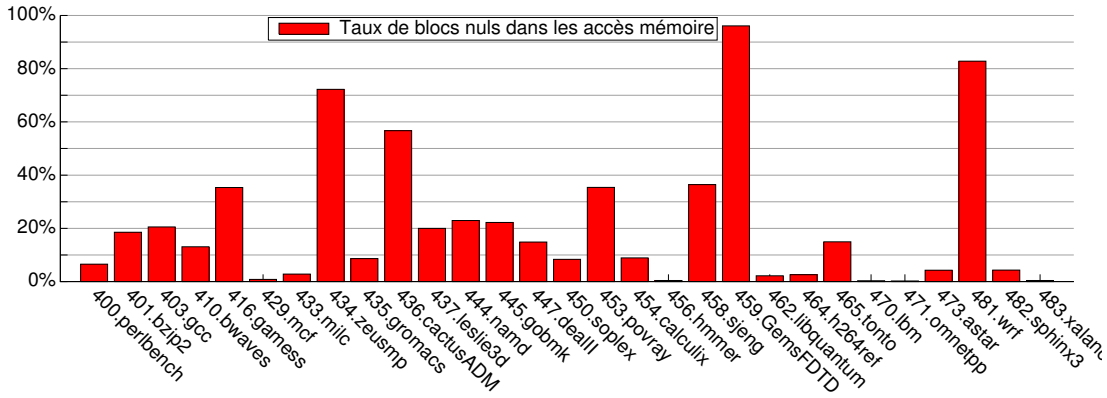


Figure 4.10 – Taux dynamique de blocs nuls

Le taux dynamique de blocs nuls est présenté figure 4.10. Il est défini comme la proportion d'accès à des blocs nuls. Il est souvent proche du taux statique de blocs nuls (figure 4.1 page 80), mais pour quelques applications il est franchement différent.

Par exemple, l'application *473.astar* affiche un taux statique de blocs nuls de 51% (figure 4.1). La *Decoupled Zero-Compressed Memory* en tire parti : ainsi 192 Mo suffisent à faire tenir une application nécessitant 384 Mo avec une mémoire non-compressée (figure 4.15 page 105). Par contre, le taux d'accès dynamique (figure 4.10) est faible : la latence est proche de celle d'une mémoire non-compressée (figure 4.9).

Les applications *429.mcf*, *471.omnetpp* et *483.xalancbmk* ont de faibles taux de blocs nuls, qu'ils soient statiques ou dynamiques. Leurs latences mémoire sont donc augmentées par la *Decoupled Zero-Compressed Memory*.

L'application *458.sjeng* a un taux dynamique de blocs nuls de 38% (figure 4.10). Pourtant, sa latence mémoire augmente de 10% (figure 4.8). Cela est dû à un nombre très important d'échecs dans le cache de descripteurs de pages. Environ 50% des accès sont des échecs.

Une façon simple de diminuer cette pénalité serait de désactiver à l'exécution la compression lorsque le taux d'échec dans le cache des descripteurs de pages est trop important.

4.2.2.4 Conclusions

L'utilisation de la *Decoupled Zero-Compressed Memory* permet de réduire la quantité de mémoire nécessaire pour contenir le *working-set* de nombreuses applications des suites SPEC CPU 2000 et 2006. Cette réduction atteint généralement un facteur compris entre 1 et 4. Pour quelques applications, ce facteur est même supérieur à 6.

Contrairement aux propositions précédentes de mémoires compressées, la *Decoupled Zero-Compressed Memory* permet aussi de réduire la latence mémoire dans de nombreux cas. C'est le cas pour la majorité des applications testées. Comme le *Zero Content Augmented Cache* présenté au chapitre 3, la *Decoupled Zero-Compressed Memory* utilise les accès dynamiques à des blocs nuls.

4.3 Mémoire découplée avec compression FPC

La *Decoupled Zero-Compressed Memory* présentée précédemment permet de compresser les blocs nuls présents en mémoire. Cependant, nous avons mesuré (figure 4.1) la présence de blocs non-nuls mais compressibles avec l'algorithme de compression FPC [8].

Les blocs non-nuls qui une fois compressés occupent moins de la moitié de leur taille initiale peuvent être stockés compressés. Nous allons présenter un mécanisme très proche de celui décrit pour la *Decoupled Zero-Compressed Memory* permettant l'utilisation de FPC. Une seule taille de bloc compressé est considérée ; en gérer plusieurs aurait un coût matériel trop important.

4.3.1 Architecture

La structure de cette nouvelle mémoire compressée utilisant un algorithme de compression dérive de la *Decoupled Zero-Compressed Memory*. La page est ici aussi découpée en blocs de B octets, mais les lignes de l'espace PC ne font plus que $B/2$ octets. Un bloc occupera donc 0, 1 ou 2 lignes selon qu'il est respectivement, nul, compressé ou non-compressé. Les 2 lignes occupées par un bloc non-compressé doivent être dans le même *set* de l'espace PC.

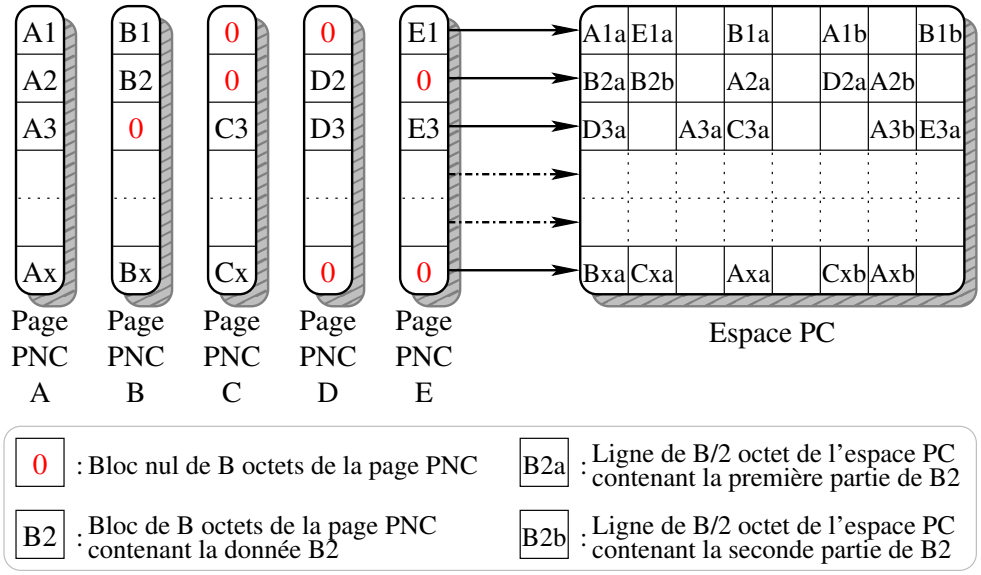


Figure 4.11 – Cinq pages Physiques Non-Compressées (pages PNC) sont affectées à un espace Physique Compressé (espace PC) de quatre pages. Chaque bloc non-nul est affecté à 1 ou 2 des 8 lignes mémoire possibles d'un *set*.

4.3.1.1 Coût de stockage des structures de contrôle

Les structures de contrôle nécessaires sont toujours les descripteurs de pages et les descripteurs d'espace. Leurs tailles ont presque doublé par rapport à la précédente proposition de mémoire compressée.

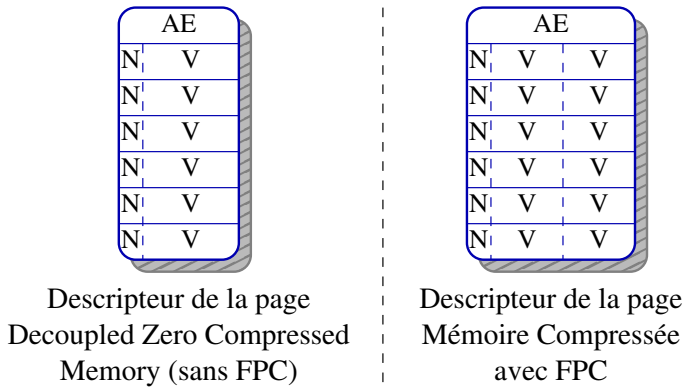


Figure 4.12 – Descripteurs de pages. À gauche celui de la *DZC Memory*, à droite celui de son extension utilisant FPC

Le descripteur de page (figure 4.12 page ci-contre) contient un pointeur d'espace PC, et pour chaque bloc de la page deux pointeurs de voies et un bit de nullité. Avec la même configuration que précédemment (des espaces PC et des pages de respectivement $S = 4$ Mo et $P = 4$ Ko), les pointeurs de voies seront de $log_2(2 \cdot \frac{S}{P}) = 11$ bits. Avec des blocs de taille $B = 64$ octets, un descripteur de page contient $\frac{P}{B} = 64$ paires de pointeurs de voies. Un pointeur d'espace de 32 bits permet d'adresser une mémoire compressée de 2^{54} octets. Cela représente $\frac{P}{B} \cdot (2 \cdot log_2(2 \cdot \frac{S}{P}) + 1) + 32 = 1472$ bits, soit 184 octets.

Le descripteur d'espace (figure 4.5 page 86) contient un bit v de validité par ligne, soit $2 \cdot \frac{S}{B} = 131072$ bits par espace PC. Les compteurs LL et MIN_{LL} sont alors sur $\lfloor log_2(2 \cdot \frac{S}{P} + 1) \rfloor = 11$ bits. Cela représente au total $2 \cdot \frac{S}{B} + (\frac{P}{B} + 1)\lfloor log_2(2 \cdot \frac{S}{P} + 1) \rfloor = 131852$ bits, soit 16481 octets.

En prenant un facteur de compression maximum de $1,5$, l'espace total requis par les structures de contrôle représente $1.5 \cdot \frac{S}{P} \cdot 184 + 16481 = 299105$ octets par espace PC, soit $7,13\%$ de la mémoire.

4.3.2 Évaluation des performances

La méthode d'évaluation des performances choisie est la même que pour la *Decoupled Zero-Compressed Memory*, présentée à la section 4.2.2.2.

La figure 4.13 page suivante présente le nombre de défauts de pages d'une mémoire non-compressée, d'une *Decoupled Zero-Compressed Memory* et d'une mémoire compressée découplée utilisant FPC. Seules sont représentées les principales [3] applications bénéficiant de la compression FPC. Ces résultats sont ici aussi très proches de ce que l'on pouvait attendre du taux statique de blocs nuls et de blocs compressibles avec FPC (figure 4.1 page 80).

Les applications contenant plus de 30% de blocs compressibles sont *403.gcc*, *445.gobmk*, *458.sjeng*, et *462.libquantum*. Ces applications montrent toutes une réduction significative de la taille de la mémoire nécessaire pour contenir le *working-set* complet.

Les applications *462.libquantum* et *471.omnetpp* n'ont que 3% de blocs nuls. Une *Decoupled Zero-Compressed Memory* n'apporte pas de gain. Cependant *462.libquantum* et *471.omnetpp* ont respectivement 45% et 19% de leurs blocs compressibles avec FPC. Ainsi, une mémoire découplée avec compression FPC permet de faire tenir dans leurs *working-sets* dans une mémoire environ 30% plus petite qu'une *Decoupled Zero-Compressed Memory* ou une mémoire non-compressée.

3. Les autres applications des SPEC CPU 2006 sont représentées en annexe dans les figures 4.14, 4.15, 4.16 et 4.17 pages 104 à 107.

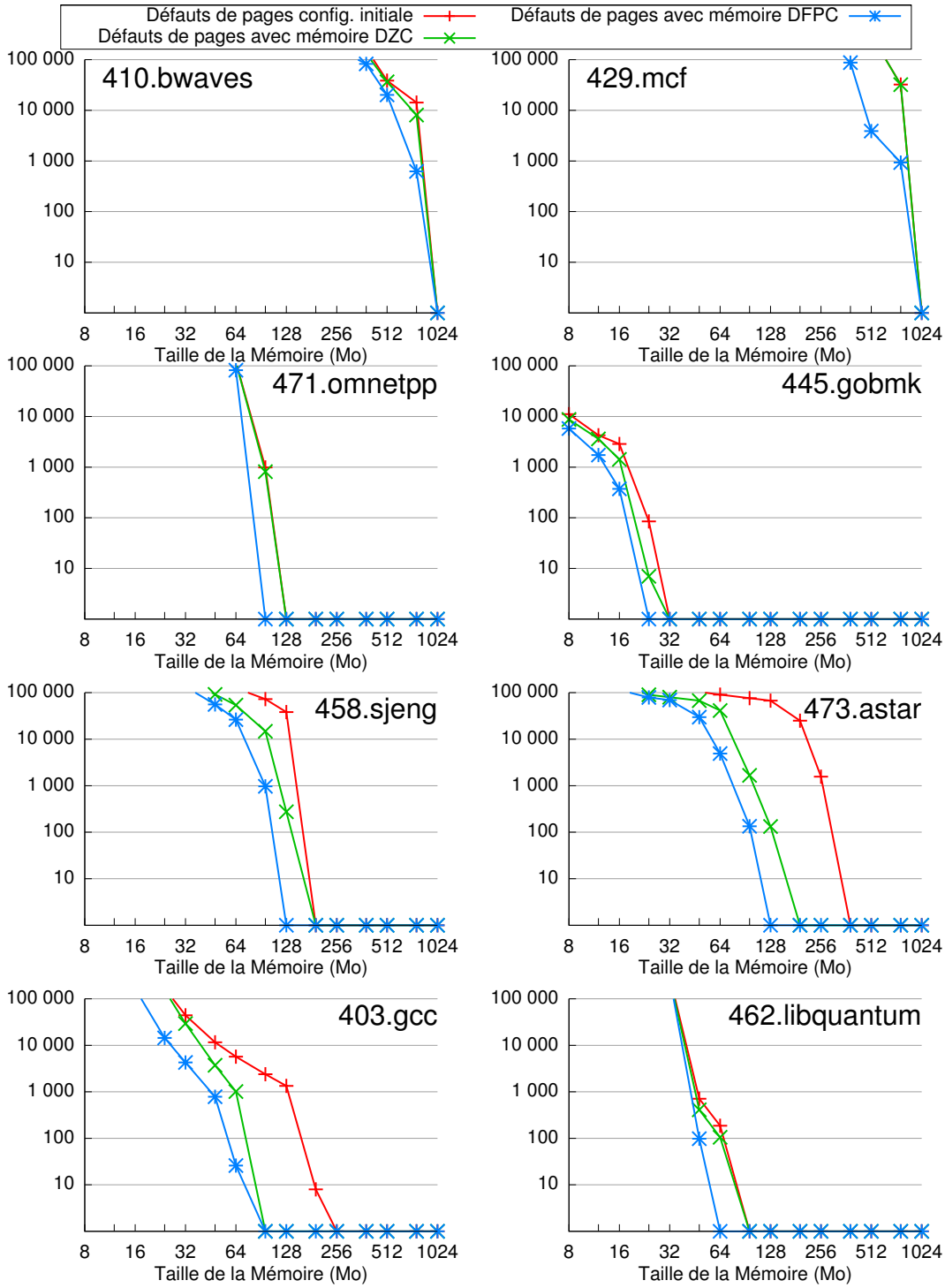


Figure 4.13 – Principales applications bénéficiant de la compression FPC

4.4 Conclusion

Dans ce chapitre, nous avons présenté deux architectures de mémoires compressées permettant une utilisation plus efficace de la mémoire physique. Notre analyse a montré que la majorité des blocs mémoire sont compressibles. Pour de nombreuses applications, une grande partie de ces blocs compressibles sont en fait des blocs nuls.

Nous avons proposé, dans un premier temps, une architecture n'exploitant que les blocs nuls. Plus de la moitié des applications en tirent un bénéfice. Leurs *working-sets* peut tenir dans une mémoire compressée plus petite qu'une mémoire non-compressée.

Nous avons ensuite proposé une architecture utilisant l'algorithme FPC. Cet algorithme très simple peut être appliqué à de petits blocs de 64 octets. Quelques applications supplémentaires utilisant peu de blocs nuls profitent elles aussi de la compression.

Les mémoires compressées que nous avons proposées permettent aussi de réduire la latence mémoire. Le gain est obtenu par un mécanisme proche du *Zero Content Augmented Cache* présenté au chapitre précédent. Le contrôleur mémoire, chargé d'effectuer la traduction d'adresse, peut en cas de lecture d'un bloc nul répondre directement sans accès à la mémoire principale. Il est à noter que le couple *Decoupled Zero-Compressed Memory* avec le *Zero Content Augmented Cache* (voir chapitre 3) permet de bénéficier d'un effet de *prefetch* et d'augmenter significativement les performances pour certaines applications.

4.5 Annexes

Dans cette section, nous présentons les évaluations pour toutes les applications de la suite SPEC CPU 2006. Le nombre de défauts de pages pour une mémoire non-compressée ainsi que pour nos deux architectures de mémoires compressées est représenté. Toutes les applications de la suite SPEC CPU 2000 présente un comportement semblable.

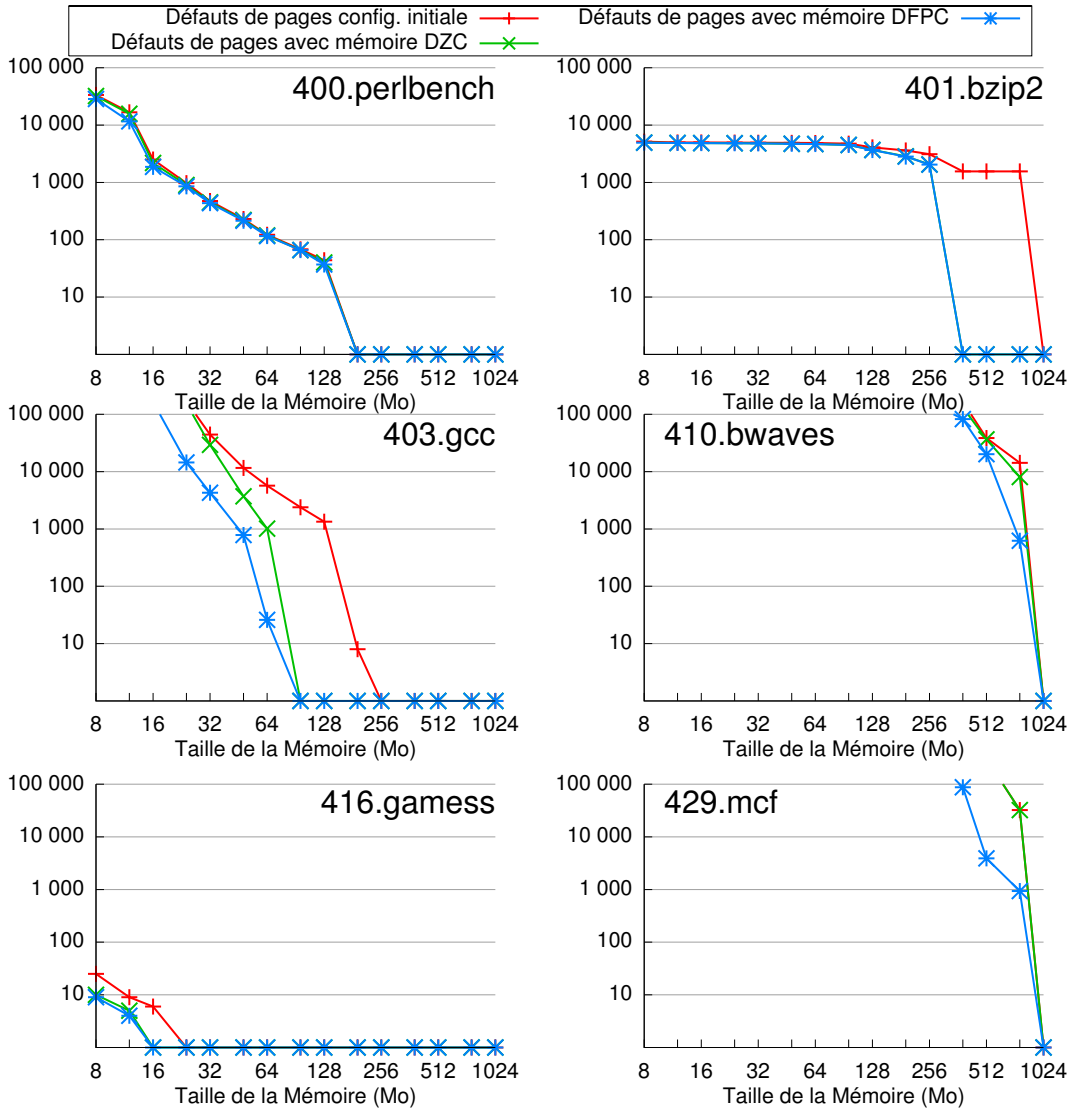


Figure 4.14 – Autres applications des SPEC CPU 2006

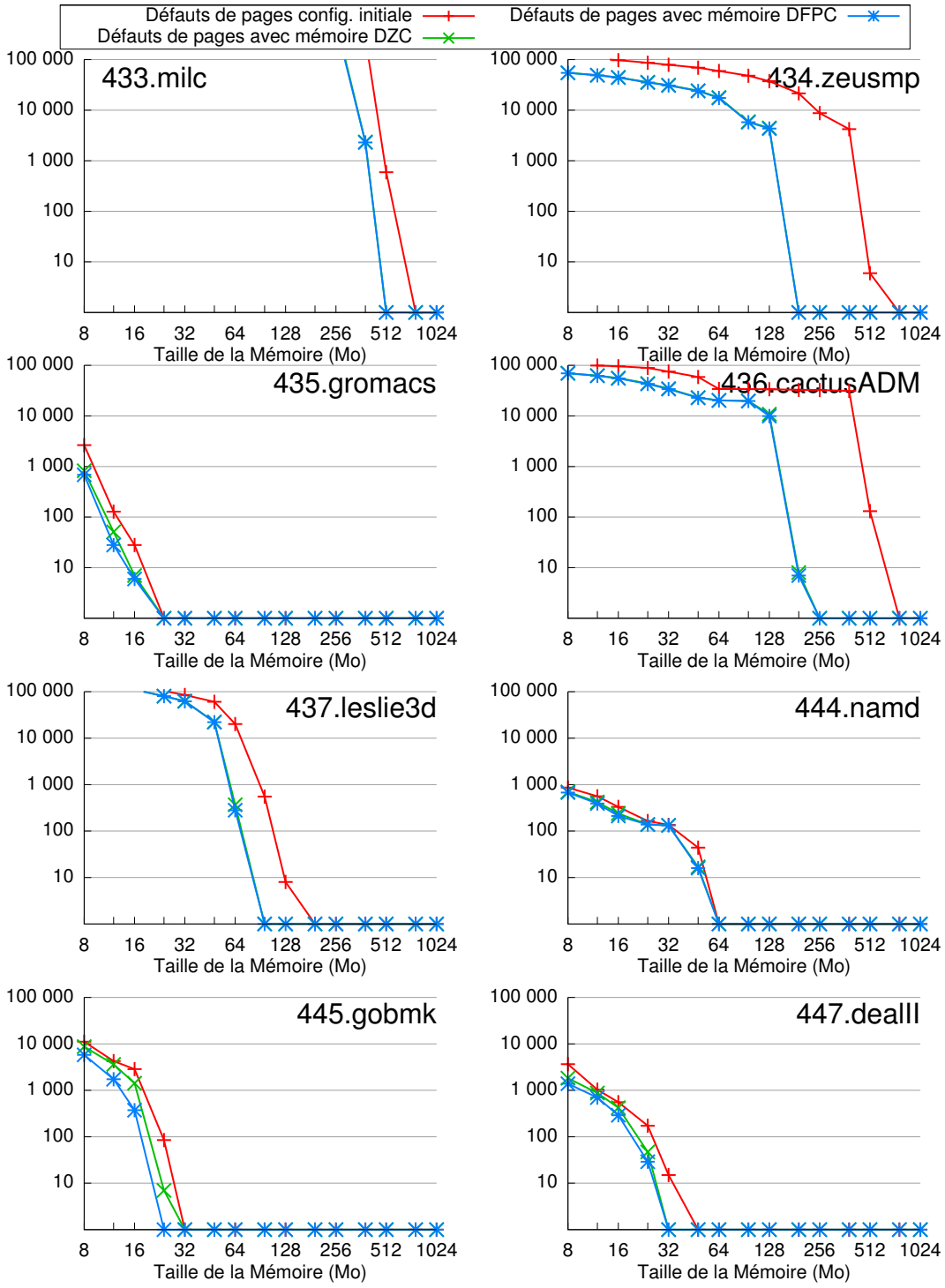


Figure 4.15 – Autres applications des SPEC CPU 2006

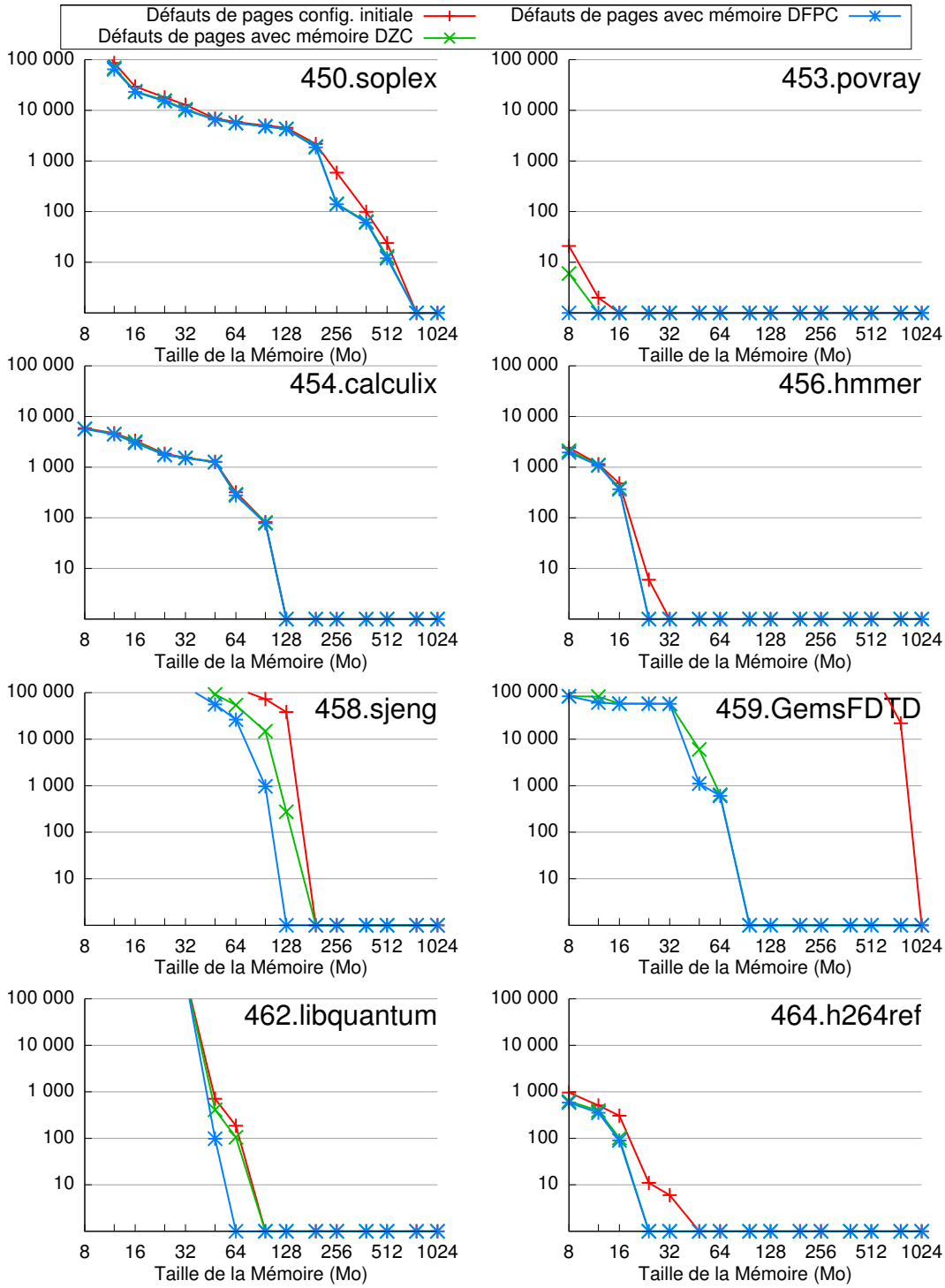


Figure 4.16 – Autres applications des SPEC CPU 2006

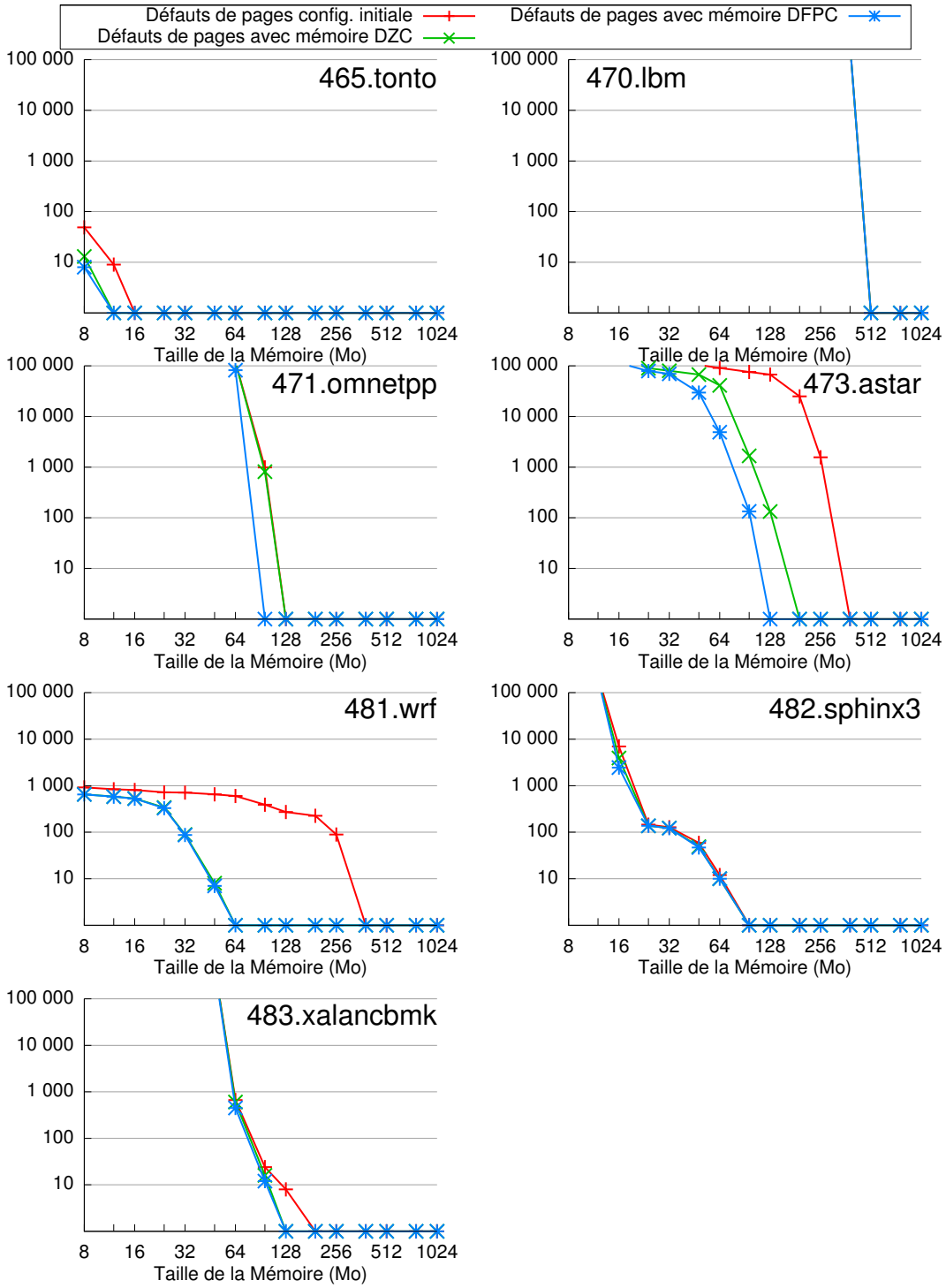


Figure 4.17 – Autres applications des SPEC CPU 2006

Conclusion

Les travaux présentés dans cette thèse montrent la présence de nombreux blocs nuls traversant les différents niveaux de la hiérarchie mémoire. Par une analyse détaillée de leurs utilisations, nous démontrons que ces blocs nuls ne sont pas de simples phénomènes d'initialisation, mais qu'ils sont présents tout au long de l'exécution de nombreuses applications. En effet, certaines applications manipulent un jeu de données partiellement nul, d'autres réinitialisent régulièrement leurs structures de contrôles. Ces nombreux blocs nuls constituent une part importante des données stockées dans les derniers niveaux de la hiérarchie mémoire.

Utiliser la hiérarchie mémoire pour transporter des blocs nuls peut être vu comme du gaspillage d'une ressource précieuse. En effet, l'augmentation incessante du nombre de cœurs, et donc du nombre de processus simultanément en cours d'exécution crée un véritable goulot d'étranglement au niveau des derniers niveaux de la hiérarchie mémoire. Les premiers niveaux sont généralement propres à un cœur d'exécution, mais les niveaux inférieurs sont partagés par un nombre toujours croissant de cœurs et donc de processus. Toute réduction du nombre d'échecs des derniers niveaux de la hiérarchie mémoire aura donc un impact important sur les performances globales du système.

Nos propositions architecturales permettent d'éliminer la majeure partie du coût de stockage et du temps d'accès aux blocs nuls pour les derniers niveaux de la hiérarchie mémoire. Nous suggérons de stocker les blocs nuls séparément des blocs non-nuls, et de ne les représenter que par un seul bit. L'espace ainsi libéré dans le cache et la mémoire principale est disponible pour les blocs non-nuls. De plus, une quantité importante de blocs nuls peut être stockée pour un coût modique. Utilisées simultanément, nos propositions permettent grâce à un mécanisme de *prefetch* d'éliminer la quasi totalité des échecs sur des blocs nuls et d'économiser la bande passante mémoire.

Bibliographie

[1] Bulent Abali, Mohammad Banikazemi, Xiawei Shen, Hubertus Franke, Dan E. Poff, et T. Basil Smith. Hardware compressed main memory : Operating system support and performance evaluation. *IEEE Transactions on Computers*, Volume 50(11), pages 1219 à 1233, Novembre 2001.
[doi>10.1109/12.966496].

[2] Bulent Abali, H. Franke, Dan E. Poff, R. A. Saccone, C. O. Schulz, L. M. Herger, et T. Basil Smith. Memory expansion technology (MXT) : Software support and performance. *IBM Journal of Research and Development*, Volume 45(2), pages 287 à 301, Mars 2001.
[doi>10.1147/rd.452.0287].

[3] Bulent Abali et Hubertus Franke. Operating system support for fast hardware compression of main memory contents. Dans *WSMWP '00 : Proceedings of the Workshop on Solving the Memory Wall Problem (in conjunction with ISCA '00)*, Vancouver, BC, Canada, Juin 2000.

[4] Bulent Abali, Hubertus Franke, Xiaowei Shen, Dan E. Poff, et T. Basil Smith. Performance of hardware compressed main memory. Dans *HPCA '01 : Proceedings of the 7th annual international symposium on High-Performance Computer Architecture*, page 73 à 81, Monterrey, NL, Mexique, Janvier 2001. IEEE Computer Society.
[doi>10.1109/HPCA.2001.903253].

[5] Advanced RISC Machines (ARM). An introduction to thumb, Mars 1995.

[6] Edward Ahn, Seung-Moon Yoo, et Sung-Mo Steve Kang. Effective algorithms for cache-level compression. Dans *GLSVLSI '01 : Proceedings of the 11th annual Great Lakes symposium on VLSI*, page 89 à 92, West Lafayette, IN, États-Unis, Avril 2001. ACM.
[doi>10.1145/368122.368872].

[7] Alaa R. Alameldeen et David A. Wood. Adaptive cache compression for high-performance processors. Dans *ISCA '04 : Proceedings of the 31st annual International Symposium on Computer Architecture*, page 212 à 223, Munich, Allemagne, Juin 2004. IEEE Computer Society.
[doi>10.1109/ISCA.2004.1310776].

[8] Alaa R. Alameldeen et David A. Wood. Frequent pattern compression : A significance-based compression scheme for L2 caches. *Technical Report 1500, Computer Sciences Department, University of Wisconsin-Madison*, Avril 2004.

[9] Guido Araùjo, Paulo Centoducatte, Rodolfo Azevedo, et Ricardo Pannain. Expression-tree-based algorithms for code compression on embedded risc architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Volume 8(5), pages 530 à 533, Octobre 2000.
[doi>10.1109/92.894158].

[10] Todd Austin, Eric Larson, et Dan Ernst. Simplescalar : An infrastructure for computer system modeling. *Computer*, Volume 35(2), pages 59 à 67, Février 2002.
[doi>10.1109/2.982917].

[11] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, Volume 5(2), pages 78 à 101, Juin 1966.
[doi>10.1147/sj.52.0078].

[12] Vicenç Beltran, Jordi Torres, et Eduard Ayguadé. Improving disk bandwidth-bound applications through main memory compression. Dans *MEDEA '07 : Proceedings of the 2007 workshop on MEmory performance : DEaling with Applications, systems and architecture (in conjunction with PACT '07)*, page 57 à 63, Braşov, Romanie, Septembre 2007. ACM.
[doi>10.1145/1327171.1327178].

[13] Vicenç Beltran, Jordi Torres, et Eduard Ayguadé. Improving web server performance through main memory compression. Dans *ICPADS '08 : Proceedings of the 14th International Conference on Parallel and Distributed Systems*, page 303 à 310, Melbourne, VIC, Australie, Décembre 2008. IEEE Computer Society.
[doi>10.1109/ICPADS.2008.15].

[14] Luca Benini, Davide Bruni, Alberto Macii, et Enrico Macii. Hardware-assisted data compression for energy minimization in systems with embedded processors. Dans *DATE '02 : Proceedings of Design, Automation and Test in Europe conference and exhibition*, page 449 à 453, Paris, France, Mars 2002. IEEE Computer Society.
[doi>10.1109/DATE.2002.998312].

[15] Luca Benini, Davide Bruni, Alberto Macii, et Enrico Macii. Memory energy minimization by data compression : algorithms, architectures and implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Volume 12(3), pages 255 à 268, Mars 2004. [doi>10.1109/TVLSI.2004.824304].

[16] Luca Benini, Alberto Macii, Enrico Macii, et Massimo Poncino. Selective instruction compression for memory energy reduction in embedded systems. Dans *ISLPED '99 : Proceedings of the 1999 international symposium on Low power electronics and design*, page 206 à 211, San Diego, CA, États-Unis, Août 1999. ACM. [doi>10.1145/313817.313927].

[17] Luca Benini, Alberto Macii, et Alberto Nannarelli. Cached-code compression for energy minimization in embedded processors. Dans *ISLPED '01 : Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, page 322 à 327, Huntington Beach, CA, États-Unis, Août 2001. ACM. [doi>10.1145/383082.383177].

[18] Luca Benini, Francesco Menichelli, et Mauro Olivieri. A class of code compression schemes for reducing power consumption in embedded microprocessor systems. *IEEE Transactions on Computers*, Volume 53(4), pages 467 à 482, Avril 2004. [doi>10.1109/TC.2004.1268405].

[19] Mauricio Jr. Breternitz et Roger Smith. Enhanced compression techniques to simplify program decompression and execution. Dans *ICCD '97 : Proceedings of the 1997 International Conference on Computer Design*, page 170, Austin, TX, États-Unis, Octobre 1997. IEEE Computer Society. [doi>10.1109/ICCD.1997.628865].

[20] Michael Burrows et David J. Wheeler. A block-sorting lossless data compression algorithm. *Research Report 124*, 1994. Digital Equipment Corporation, Palo Alto, CA, États-Unis.

[21] Ramon Canal, Antonio González, et James E. Smith. Very low power pipelines using significance compression. Dans *MICRO 33 : Proceedings of the 33rd annual international symposium on Microarchitecture*, page 181 à 190, Monterey, CA, États-Unis, Décembre 2000. ACM. [doi>10.1145/360128.360147].

[22] R. Cervera, T. Cortes, et Y. Becerra. Improving application performance through swap compression. Dans *ATEC '99 : Proceedings of the annual conference on USENIX Annual Technical Conference*, page 46 à 58, Monterey, CA, États-Unis, Juin 1999. USENIX Association.

[23] David Chen, Enoch Peserico, et Larry Rudolph. A dynamically partitionable compressed cache. Dans *In Proceedings of the Singapore-MIT Alliance Symposium*, 2003.

[24] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, et M. Wolczko. Heap compression for memory-constrained java environments. Dans *OOPSLA '03 : Proceedings of the 18th annual conference on Object-Oriented Programing, Systems, Languages, and Applications*, page 282 à 301, Anaheim, CA, États-Unis, Octobre 2003. ACM.
[doi>10.1145/949305.949330].

[25] Daniel Citron. Exploiting low entropy to reduce wire delay. *IEEE Computer Architecture Letters*, Volume 3(1), pages 1, 2004.
[doi>10.1109/L-CA.2004.7].

[26] Daniel Citron et Larry Rudolph. Creating a wider bus using caching techniques. Dans *HPCA '95 : Proceedings of the 1st annual international symposium on High-Performance Computer Architecture*, page 90 à 99, Raleigh, NC, États-Unis, Janvier 1995. IEEE Computer Society.
[doi>10.1109/HPCA.1995.386552].

[27] Compressed caching for linux.
http://code.google.com/p/compcache/.

[28] Keith D. Cooper et Nathaniel McIntosh. Enhanced code compression for embedded risc processors. Dans *PLDI '99 : Proceedings of the 1999 international conference on Programming Language Design and Implementation*, page 139 à 149, Atlanta, GA, États-Unis, Mai 1999. ACM.
[doi>10.1145/301618.301655].

[29] Rodrigo S. de Castro, Alair Pereira do Lago, et Dilma Da Silva. Adaptive compressed caching : Design and implementation. Dans *SBAC-PAD '03 : Proceedings of the 15th Symposium on Computer Architecture and High Performance Computing*, page 10 à 18, São Paulo, SP, Brésil, Novembre 2003. IEEE Computer Society.
[doi>10.1109/CAHPC.2003.1250316].

[30] Fred Douglis. The compression cache : Using on-line compression to extend physical memory. Dans *USENIX Winter : Proceedings of 1993 Winter USENIX Conference*, page 519 à 529, San Diego, CA, États-Unis, Janvier 1993. USENIX Association.

[31] Julien Dusser, Thomas Piquet, et André Seznec. Zero-content augmented caches. Dans *ICS '09 : Proceedings of the 23rd annual International Conference on Supercomputing*, page 46 à 55, Yorktown Heights, NY, États-Unis, Juin 2009. ACM.
[doi>10.1145/1542275.1542288].

[32] Julien Dusser et André Seznec. Decoupled zero-compressed memory. Dans *HIPEAC '11 : Proceedings of the 6th annual international conference on High Performance Embedded Architectures and Compilers*, Heraklion, Crete, Grèce, Janvier 2011. ACM.

[33] Magnus Ekman et Per Stenström. A cost-effective main memory organization for future servers. Dans *IPDPS '05 : Proceedings of the 19th International Parallel and Distributed Processing Symposium*, page 45 à 55, Denver, CO, États-Unis, Avril 2005. IEEE Computer Society.
[doi>10.1109/IPDPS.2005.12].

[34] Magnus Ekman et Per Stenström. A robust main-memory compression scheme. Dans *ISCA '05 : Proceedings of the 32nd annual International Symposium on Computer Architecture*, page 74 à 85, Madison, WI, États-Unis, Juin 2005. IEEE Computer Society.
[doi>10.1109/ISCA.2005.6].

[35] Jens Ernst, William Evans, Christopher W. Fraser, Todd A. Proebsting, et Steven Lucco. Code compression. *SIGPLAN Notice*, Volume 32(5), pages 358 à 365, Mai 1997.
[doi>10.1145/258916.258947].

[36] ext4 - extended file system 4, kernelbies documentation.
http://kernelnewbies.org/Ext4.

[37] Robert M. Fano. Transmission of information. *Technical Report No. 65*, 1949. M.I.T. Research Laboratory of Electronics, Cambridge, MA, États-Unis.

[38] Matthew Farrens et Arvin Park. Dynamic base register caching : A technique for reducing address bus width. Dans *ISCA '91 : Proceedings of the 18th annual International Symposium on Computer Architecture*, page 128 à 137, Toronto, ON, Canada, Mai 1991. IEEE Computer Society.
[doi>10.1109/ISCA.1991.1021606].

[39] Peter A. Franaszek, Philip Heidelberger, Dan E. Poff, et John T. Robinson. Algorithms and data structures for compressed-memory machines. *IBM Journal of Research and Development*, Volume 45(2), pages 245 à 258, Mars 2001.
[doi>10.1147/rd.452.0245].

[40] Peter A. Franaszek, Philip Heidelberger, et Michael Wazlowski. On management of free space in compressed memory systems. Dans *SIGMETRICS '99 : Proceedings of the 1999 international conference on measurement and modeling of computer systems*, page 113 à 121, Atlanta, GA, États-Unis, Juin 1999. ACM.
[doi>10.1145/301453.301485].

[41] Peter A. Franaszek et John T. Robinson. Design and analysis of internal organizations for compressed random access memories. *Technical Report RC 21146, IBM T. J. Watson Research Center*, Octobre 1998.

[42] Peter A. Franaszek et John T. Robinson. On internal organization in compressed random-access memories. *IBM Journal of Research and Development*, Volume 45(2), pages 259 à 270, Mars 2001.
[doi>10.1147/rd.452.0259].

[43] Peter A. Franaszek, John T. Robinson, et J. Thomas. Parallel compression with cooperative dictionary construction. Dans *DCC '96 : Proceedings of the 6th annual Data Compression Conference*, page 200 à 209, Snowbird, UT, États-Unis, Mars 1996. IEEE Computer Society.
[doi>10.1109/DCC.1996.488325].

[44] Christopher W. Fraser. Automatic inference of models for statistical code compression. Dans *PLDI '99 : Proceedings of the 1999 international conference on Programming Language Design and Implementation*, page 242 à 246, Atlanta, GA, États-Unis, Mai 1999. ACM.
[doi>10.1145/301618.301672].

[45] Christopher W. Fraser, Eugene W. Myers, et Alan L. Wendt. Analyzing and compressing assembly code. Dans *SIGPLAN '84 : Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, page 117 à 121, Montréal, Canada, Juin 1984. ACM.
[doi>10.1145/502874.502886].

[46] Freddy Gabbay. Speculative execution based on value prediction. *Technical Report No. 1080*, 1996. Electrical Engineering Department, Technion, Israel Institue of Technology.

[47] Erik G. Hallnor et Steven K. Reinhardt. A compressed memory hierarchy using an indirect index cache. Dans *WMPI '04 : Proceedings of the 3rd Workshop on Memory Performance Issues (in conjunction with ISCA '04)*, page 9 à 15, Munich, Allemagne, Juin 2004. ACM.
[doi>10.1145/1054943.1054945].

[48] Erik G. Hallnor et Steven K. Reinhardt. A unified compressed memory hierarchy. Dans *HPCA '05 : Proceedings of the 11th annual international symposium on High-Performance Computer Architecture*, page 201 à 212, San Francisco, CA, États-Unis, Février 2005. IEEE Computer Society.
[doi>10.1109/HPCA.2005.4].

[49] Douglas R. Hofstadter. *Gödel, Escher, Bach : an Eternal Golden Braid*. Basic Books, 1979.

[50] Jan Hoogerbrugge, Lex Augusteijn, Jeroen Trum, et Rik Van De Wiel. A code compression system based on pipelined interpreters. *Software -- Practice and Experience*, Volume 29(11), pages 1005 à 2023, Septembre 1999. [doi>10.1002/(SICI)1097-024X(199909)29:11<1005::AID-SPE270>-3.0.CO;2-F].

[51] David A. Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the Institute of Radio Engineers*, Volume 40(9), pages 1098 à 1101, Septembre 1952. [doi>10.1109/JRPROC.1952.273898].

[52] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A : Instruction Set Reference, A-M*, Juin 2010.

[53] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B : Instruction Set Reference, N-Z*, Juin 2010.

[54] Mafijul Md. Islam et Per Stenström. Zero-value caches : Cancelling loads that return zero. Dans *PACT '09 : Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, page 237 à 245, Raleigh, NC, États-Unis, Septembre 2009. IEEE Computer Society. [doi>10.1109/PACT.2009.29].

[55] Krishna Kant et Ravi Iyer. Compressibility characteristics of address/data transfers in commercial workloads. Dans *CAECW '02 : Proceedings of the 5th Workshop on Computer Architecture Evaluation Using Commercial Workloads (in conjunction with HPCA '02)*, Cambridge, MA, États-Unis, Septembre 2002.

[56] M. Kjelsø, M. Gooch, et S. Jones. Design and performance of a main memory hardware data compressor. Dans *Euromicro '96 : Proceedings of the 22nd annual Euromicro Conference*, page 423 à 430, Prague, République tchèque, Septembre 1996. IEEE Computer Society. [doi>10.1109/EURMIC.1996.546466].

[57] David Kroft. Lockup-free instruction fetch/prefetch cache organization. Dans *ISCA '81 : Proceedings of the 8th annual International Symposium on Computer Architecture*, page 81 à 87, Minneapolis, MN, États-Unis, Mai 1981. IEEE Computer Society. [doi>10.1145/285930.285979].

[58] Jang-Soo Lee, Won-Kee Hong, et Shin-Dug Kim. Design and evaluation of a selective compressed memory system. Dans *ICCD '99 : Proceedings of the 1999 International Conference on Computer Design*, page 184, Austin, TX, États-Unis, Octobre 1999. IEEE Computer Society.
[doi>10.1109/ICCD.1999.808424].

[59] Jang-Soo Lee, Won-Kee Hong, et Shin-Dug Kim. A selective compressed memory system by on-line data decompressing. Dans *Euromicro '99 : Proceedings of the 25th annual Euromicro Conference*, volume 1, page 224 à 227, Milan, Italie, Septembre 1999. IEEE Computer Society.
[doi>10.1109/EURMIC.1999.794470].

[60] Charles Lefurgy, Eva Piccininni, et Trevor Mudge. Reducing code size with run-time decompression. Dans *HPCA '00 : Proceedings of the 6th annual international symposium on High-Performance Computer Architecture*, page 218 à 228, Toulouse, France, Janvier 2000.
[doi>10.1109/HPCA.2000.824352].

[61] Haris Lekatsas, Jörg Henkel, et Wayne Wolf. Code compression for low power embedded system design. Dans *DAC '00 : Proceedings of the 37th annual Design Automation Conference*, page 294 à 299, Los Angeles, CA, États-Unis, Juin 2000. ACM.
[doi>10.1145/337292.337423].

[62] Haris Lekatsas et Wayne Wolf. Code compression for embedded systems. Dans *DAC '98 : Proceedings of the 35th annual Design Automation Conference*, page 516 à 521, San Francisco, CA, États-Unis, Juin 1998. ACM.
[doi>10.1145/277044.277185].

[63] Kevin M. Lepak et Mikko H. Lipasti. Silent stores for free. Dans *MICRO 33 : Proceedings of the 33rd annual international symposium on Microarchitecture*, page 22 à 31, Monterey, CA, États-Unis, Décembre 2000. ACM.
[doi>10.1145/360128.360133].

[64] Mikko H. Lipasti, Christopher B. Wilkerson, et John Paul Shen. Value locality and load value prediction. Dans *ASPLOS-VII : Proceedings of the 7th annual international conference on Architectural Support for Programming Languages and Operating Systems*, page 138 à 147, Cambridge, MA, États-Unis, Octobre 1996. ACM.
[doi>10.1145/237090.237173].

[65] MIPS Technologies. *MIPS32®Architecture For Programmers, Volume II-A : The MIPS32®Instruction Set*, Mars 2010.

[66] Keith E. Moore. Compressing memory management in a device, Mai 2003. U.S. patent 6564305, Hewlett-Packard Development Company.

[67] Robert Michael Muth. *Alto : a platform for object code modification.* PhD thesis, The University of Arizona, 1999. Director : Saumya K. Debray.

[68] NTFS - new technology file system.
http://www.ntfs.com/.

[69] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, et Brad Calder. Using simpoint for accurate and efficient simulation. Dans *SIGMETRICS '03 : Proceedings of the 2003 international conference on measurement and modeling of computer systems*, page 318 à 319, San Diego, CA, États-Unis, Juin 2003. ACM.
[doi>10.1145/781027.781076].

[70] Prateek Pujara et Aneesh Aggarwal. Increasing cache capacity through word filtering. Dans *ICS '07 : Proceedings of the 21st annual International Conference on Supercomputing*, page 222 à 231, Seattle, WA, États-Unis, Juin 2007. ACM.
[doi>10.1145/1274971.1275002].

[71] Moinuddin K. Qureshi, M. Aater Suleman, et Yale N. Patt. Line distillation : Increasing cache capacity by filtering unused words in cache lines. Dans *HPCA '07 : Proceedings of the 13th annual international symposium on High-Performance Computer Architecture*, page 250 à 259, Phoenix, AZ, États-Unis, Février 2007. IEEE Computer Society.
[doi>10.1109/HPCA.2007.346202].

[72] Hans Reiser. Reiser4 is released ! (disponible sur archive.org).
http://www.namesys.com/v4/v4.html.

[73] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, et Pablo Montesinos. SESC simulator, 2005.
http://sesc.sourceforge.net.

[74] Luigi Rizzo. A very fast algorithm for ram compression. *SIGOPS Operating Systems Review*, Volume 31(2), pages 36 à 45, Avril 1997.
[doi>10.1145/250007.250012].

[75] Sumit Roy, Raj Kumar, et Milos Prvulovic. Improving system performance with compressed memory. Dans *IPDPS '01 : Proceedings of the 15th International Parallel and Distributed Processing Symposium*, page 66 à 71, San Francisco, CA, États-Unis, Avril 2001. IEEE Computer Society.
[doi>10.1109/IPDPS.2001.925011].

[76] Jennifer B. Sartor, Stephen M. Blackburn, Daniel Frampton, Martin Hirzel, et Kathryn S. McKinley. Z-rays : Divide arrays and conquer speed and flexibility. Dans *PLDI '10 : Proceedings of the 2010 international conference on Programming Language Design and Implementation*, page 471 à 482, Toronto, ON, Canada, Juin 2010. ACM.
[doi>10.1145/1806596.1806649].

[77] Jennifer B. Sartor, Martin Hirzel, et Kathryn S. McKinley. No bit left behind : the limits of heap data compression. Dans *ISMM '08 : Proceedings of the 7th annual International Symposium on Memory Management*, page 111 à 120, Tucson, AZ, États-Unis, Juin 2008. ACM.
[doi>10.1145/1375634.1375651].

[78] Christoph Scheurich et Michel Dubois. The design of a lockup-free cache for high-performance multiprocessors. Dans *SC '88 : Proceedings of the 1988 international conference on Supercomputing*, page 352 à 359, Orlando, FL, États-Unis, Novembre 1988.
[doi>10.1109/SUPERC.1988.44672].

[79] André Seznec. A case for two-way skewed-associative caches. Dans *ISCA '93 : Proceedings of the 20th annual International Symposium on Computer Architecture*, page 169 à 178, San Diego, CA, États-Unis, Mai 1993. ACM.
[doi>10.1145/165123.165152].

[80] André Seznec. Decoupled sectored caches : conciliating low tag implementation cost. Dans *ISCA '94 : Proceedings of the 21st annual International Symposium on Computer Architecture*, page 384 à 393, Chicago, IL, États-Unis, Avril 1994. IEEE Computer Society.
[doi>10.1109/ISCA.1994.288133].

[81] André Seznec. Concurrent support of multiple page sizes on a skewed associative TLB. *IEEE Transactions on Computers*, Volume 53(7), pages 924 à 927, Juillet 2004.
[doi>10.1109/TC.2004.21].

[82] André Seznec. Analysis of the o-geometric history length branch predictor. Dans *ISCA '05 : Proceedings of the 32nd annual International Symposium on Computer Architecture*, page 394 à 405, Madison, WI, États-Unis, Juin 2005. IEEE Computer Society.
[doi>10.1109/ISCA.2005.13].

[83] Claude Elwood Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, Volume 27, pages 379--423, Juillet 1948.

[84] T. Basil Smith, Bulent Abali, Dan E. Poff, et R. Brett Tremaine. Memory expansion technology (MXT) : Competitive impact. *IBM Journal of Research and Development*, Volume 45(2), pages 303 à 309, Mars 2001.
[doi>10.1147/rd.452.0303].

[85] James A. Storer et Thomas G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, Volume 29(4), pages 928 à 951, Octobre 1982.
[doi>10.1145/322344.322346].

[86] Madhusudhan Talluri et Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. Dans *ASPLOS-VI : Proceedings of the 6th annual international conference on Architectural Support for Programming Languages and Operating Systems*, page 171 à 182, San José, CA, États-Unis, Décembre 1994. ACM.
[doi>10.1145/195473.195531].

[87] Madhusudhan Talluri, Shing Kong, Mark D. Hill, et David A. Patterson. Tradeoffs in supporting two page sizes. Dans *ISCA '92 : Proceedings of the 19th annual International Symposium on Computer Architecture*, page 415 à 424, Gold Coast, QLD, Australie, Mai 1992. ACM.
[doi>10.1145/139669.140406].

[88] David Tarjan, Shyamkumar Thoziyoor, et Norman P. Jouppi. Cacti 4.2.
http://quid.hpl.hp.com:9081/cacti/.

[89] Xinhua Tian et Minxuan Zhang. A unified compressed cache hierarchy using simple frequent pattern compression and partial cache line prefetching. Dans *ICESS '07 : Proceedings of the 3rd international conference on Embedded Software and Systems*, page 142 à 153, Daegu, Corée du sud, Mai 2007. Springer-Verlag.
[doi>10.1007/978-3-540-72685-2_14].

[90] R. Brett Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. Basil Smith, Mike E. Wazlowski, et P. M. Bland. IBM memory eXpansion technology (MXT). *IBM Journal of Research and Development*, Volume 45(2), pages 271 à 285, Mars 2001.
[doi>10.1147/rd.452.0271].

[91] Irina Chihaia Tuduce et Thomas Gross. Adaptive main memory compression. Dans *ATEC '05 : Proceedings of the annual conference on USENIX Annual Technical Conference*, page 237 à 250, Anaheim, CA, États-Unis, Avril 2005. USENIX Association.

[92] Luis Villa, Michael Zhang, et Krste Asanović. Dynamic zero compression for cache energy reduction. Dans *MICRO 33 : Proceedings of the 33rd annual international symposium on Microarchitecture*, page 214 à 220, Monterey, CA, États-Unis, Décembre 2000. ACM.
[doi>10.1145/360128.360150].

[93] Terry A. Welch. A technique for high-performance data compression. *IEEE Computer*, Volume 17(6), pages 8 à 19, Juin 1984.
[doi>10.1109/MC.1984.1659158].

[94] Ross N. Williams. An extremely fast ziv-lempel data compression algorithm. Dans *DCC '91 : Proceedings of the 1st annual Data Compression Conference*, page 362 à 371, Snowbird, UT , États-Unis, Avril 1991.
[doi>10.1109/DCC.1991.213344].

[95] Paul R. Wilson. Operating system support for small objects. Dans *IWOOOS '91 : Proceedings of the 1st International Workshop on Object Orientation in Operating Systems*, page 80 à 86, Palo Alto, CA, États-Unis, Octobre 1991. IEEE Computer Society.
[doi>10.1109/IWOOOS.1991.183026].

[96] Paul R. Wilson, Scott F. Kaplan, et Yannis Smaragdakis. The case for compressed caching in virtual memory systems. Dans *ATEC '99 : Proceedings of the annual conference on USENIX Annual Technical Conference*, page 101 à 116, Monterey, CA, États-Unis, Juin 1999. USENIX Association.

[97] Andrew Wolfe et Alex Chanin. Executing compressed programs on an embedded risc architecture. *SIGMICRO Newsletter*, Volume 23(1-2), pages 81 à 91, Décembre 1992.
[doi>10.1145/144965.145003].

[98] David A. Wood et Alaa R. Alameldeen. Adaptive cache compression system, Mars 2005. U.S. patent 7412564, Wisconsin Alumni Research Foundation.

[99] Xianhong Xu, Christopher T. Clarke, et Simon R. Jones. High performance code compression architecture for the embedded arm/thumb processor. Dans *CF '04 : Proceedings of the 1st annual conference on Computing Frontiers*, page 451 à 456, Ischia, Italie, Avril 2004. ACM.
[doi>10.1145/977091.977154].

[100] Jun Yang et Rajiv Gupta. Frequent value locality and its applications. *ACM Transactions on Embedded Computing Systems*, Volume 1(1), pages 79 à 105, Novembre 2002.
[doi>10.1145/581888.581894].

[101] Jun Yang, Youtao Zhang, et Rajiv Gupta. Frequent value compression in data caches. Dans *MICRO 33 : Proceedings of the 33rd annual international symposium on Microarchitecture*, page 258 à 265, Monterey, CA, États-Unis, Décembre 2000. ACM.
[doi>10.1145/360128.360154].

[102] Lei Yang, Haris Lekatsas, et Robert P. Dick. High-performance operating system controlled memory compression. Dans *DAC '06 : Proceedings of the 43rd annual Design Automation Conference*, page 701 à 704, San Francisco, CA, États-Unis, Juillet 2006. ACM.
[doi>10.1145/1146909.1147086].

[103] Youtao Zhang, Jun Yang, et Rajiv Gupta. Frequent value locality and value-centric data cache design. Dans *ASPLOS-IX : Proceedings of the 9th annual international conference on Architectural Support for Programming Languages and Operating Systems*, page 150 à 159, Cambridge, MA, États-Unis, Novembre 2000. ACM.
[doi>10.1145/378993.379235].

[104] Jacob Ziv et Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, Volume 23(3), pages 337 à 343, Mai 1977.

[105] Jacob Ziv et Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, Volume 24(5), pages 530 à 536, Septembre 1978.

Table des figures

Résumé

La hiérarchie mémoire subit une pression qui ne cesse de croître. Cette pression a eu pour origine la montée en fréquence des processeurs. Cependant, maintenant que la fréquence stagne autour de 3 GHz, le nombre de cœurs d'exécution et donc le nombre de processus s'exécutant simultanément augmentent à leur tour. La hiérarchie mémoire subit alors un nombre croissant de requêtes, conduisant à la saturation de sa bande passante.

Les travaux présentés dans cette thèse montrent que la hiérarchie mémoire est souvent utilisée pour transporter des blocs de données totalement nuls. Ces blocs de valeur triviale se trouvent particulièrement nombreux au dernier niveau de cache et au niveau de la mémoire principale. Nous proposons dans ce document d'utiliser un cache spécialisé dans la gestion de ces blocs nuls, le *Zero-Content Augmented Cache*. Ce dernier est composé d'un cache traditionnel et d'un cache dédié aux blocs nuls. Cette proposition permet à la fois d'augmenter les performances globales du système et de réduire significativement la bande passante mémoire utilisée. Dans ce document, nous proposons également une architecture de mémoire compressée utilisant la présence de blocs nuls, la *Decoupled Zero-Compressed Memory*. Cette mémoire permet de stocker un *working-set* plus grand que la taille de la mémoire physique, et donc de réduire significativement le nombre d'accès aux périphériques de stockage de masse.

Abstract

The memory hierarchy undergoes a growing pressure. This pressure has been due to the increasing frequency of the processors. However, now that the frequency stays around 3 GHz, the number of execution cores and thus the number of processes running simultaneously are increasing. The growing number of requests handled by the memory hierarchy leads to bandwidth saturation.

This study shows that the memory hierarchy is often used to transport null data blocks. These trivial value blocks are particularly numerous in the last level cache and in the main memory. We propose in this thesis to use the *Zero-Content Augmented Cache*, a cache specialized in the management of these null blocks. It consists of a traditional cache and a cache dedicated to null blocks. This proposal allows increasing overall system performance and significantly reducing memory bandwidth usage. In this document, we also propose the *Decoupled Zero-Compressed Memory*, a compressed memory architecture also using the null blocks. This compressed memory can store a working-set greater than the size of the physical memory, and thus significantly reduce the number of accesses to the mass storage devices.